VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky

DIPLOMOVÁ PRÁCE

2012 Marek Říhošek

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Grafický editor ontologií
Graphical Ontology Editor

2012 Marek Říhošek

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Marek Říhošek**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Grafický editor ontologií
Graphical Ontology Editor

Zásady pro vypracování:

Cílem této diplomové práce je vytvořit framework pro možnost sestavení vlastního grafického editoru ontologií. Zejména pak např. ontologie softwarových procesů, modelování byznys procesů apod.
Zápis ontologií v současných nástrojích je komplikovaný, neumožňuje využít výhody tohoto způsobu popisu. Cílem práce je vytvořit framework, který podpoří jednoduché vytváření vlastních grafických editorů, které zjednoduší vytváření ontologií.
Práce je určena pro více studentů, kteří budou spolupracovat na tvorbě celého frameworku.
Framework bude obsahovat zejména tyto části:
1. Vytvoření jádra prototypové architektury pro daný framework.
2. Vytvoření možnosti vygenerování si vlastního grafického nástroje např. určením způsobu zobrazování základních elementů a typů vazeb a následném vygenerování nástroje.

Na příkladu ontologií pro softwarové procesy , prozatím zachycení jen statického aspektu procesů, ověřit funkčnost frameworku.

Seznam doporučené odborné literatury:

Dle pokynů vedoucího diplomové práce.

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. Svatopluk Štolfa, Ph.D.**

Datum zadání: 18.11.2011
Datum odevzdání: 04.05.2012

VYSOKÁ ŠKOLA BÁŇSKÁ-TECHNICKÁ UNIVERZITA Fakulta elektrotechniky a informatiky OSTRAVA

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně.

Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě dne: 2. 5. 2012 Podpis : 

Poděkování

Rád bych na tomto místě poděkoval Ing. Svatoplukovi Štolfovi, Ph. D., za odborné vedení během vytváření diplomové práce, za připomínky a rady během vytváření výsledného Frameworku.

Abstrakt

Cílem této diplomové práce je vytvořit Framework, pomocí kterého si lze zpracovat načtenou ontologii ve vlastním grafickém editoru ontologií. Jednotlivé elementy ontologie si uživatel sám navolí, jakým grafickým prvkem se budou zobrazovat. Framework bude naprogramován v programovacím jazyce Java. Téma je obsáhlejšího charakteru a proto se na tvorbě podílelo více diplomantů. V práci jsem se zaměřil na problematiku načtení vstupní ontologie do jádra aplikace pomocí OWL API a následnou práci s načtenou ontologií. Dále jsem se podílel na tvorbě samotného GUI aplikace. Obsahem diplomové práce je jak samotný Framework, tak i textová část. V textové části je popsána problematika ontologií, dále pak popis jazyka OWL. Následuje popis vlastního řešení Frameworku pro zpracovaní ontologií. Obsahem je i testovací ontologie, která byla vytvořena k otestování správné funkcionality výsledného Frameworku.

Klíčová slova

Ontologie, Java, Eclipse, GUI, Framework, grafický editor, OWL, OWL API

The Abstract

The aim of this thesis is to create Framework, through which you can handle the loaded ontology graphic editor in its own ontology. The graphics elements of individual elements of the ontology could be defined by user by themselves. The Framework will be at Java programming language. The thesis is voluminous nature and therefore participated in the creation of more students. In my work I focused on the issue of load input ontology into the core of application using OWL API and subsequent work with the loaded ontology. I also participated in the creation of GUI application itself. The content of the thesis is the Framework itself and also the text part of thesis. The text section describes the problems of ontology, as well as a description language OWL. The following part is a description of a custom solution for processing ontology to the Framework. The content thesis is also testing ontology which was created to test the correct functionality of the final Framework.

Key words

Ontology, Java, Eclipse, GUI, Framework, graphics editor, OWL, OWL API

Obsah

1 Úvod

1.1 Motivace

Ontologické inženýrství si lze definovat jako *soubor aktivit souvisejících s procesem vývoje ontologií, jejich životním cyklem, a s metodikami, nástroji a jazyky pro tvorbu ontologií* [9]. Zaměříme-li se na informatické ontologie, na rozdíl od filozofických, které jsou výsledkem vědeckého zkoumání, můžeme říct, že jsou to artefakty, které jsou navrhovány a konstruovány. Přístup k návrhu ontologií prodělal analogický vývoj jako v případě počítačových programů a expertních systémů, jen s cca třicetiletým časovým posunem: od původního pojetí návrhu ontologie jako umění či vědy, dostupných jen několika výjimečně disponovaným jedincům, k inženýrství. Analogicky k softwarovému a znalostnímu inženýrství se zformovala nová inženýrská disciplína – ontologické inženýrství, a rovněž analogicky k CASE systémům v softwarovém inženýrství už jsou k dispozici i specializované softwarové nástroje podporující návrh ontologií (např., WebOnto[1], OntoEdit[2], Protégé [3]).

V současné době asi nejrozšířenějším nástrojem pro návrh ontologií je Protégé vyvinutý týmem M. Musena v institutu Stanford Medical Informatics. Avšak jeho nepříliš uživatelsky přívětivé používání, jak je uvedeno v kapitole 1.4.3.1, vedlo k úvaze, naprogramovat si vlastní grafický editor ontologií. Především prozkoumat možnost načtení již vytvořené ontologie do jádra aplikace, kde bude možné s touto ontologií objektově pracovat, je jednou z klíčových funkcionalit výsledného grafického editoru. Dále je kladen důraz na vytváření uživatelem definovaných pohledů nad ontologií, které si lze následně zobrazit. Takto zobrazené ontologie lze modifikovat. Všechny tyto zamýšlené vlastnosti Protégé nenabízí a tak je nasnadě si vlastní grafický editor naprogramovat a využít všech nedostatků stávajících editorů v náš prospěch.

1.2 Cíle práce

Cílem práce je naprogramovat Framework pomocí něhož si bude možné sestavit vlastní grafický editor ontologií. Výsledný editor bude zaměřen na tvorbu softwarových procesů a modelování byznys procesů [*16*]. Pomocí grafického editoru bude možné editovat libovolnou vstupní ontologii načtenou buď z lokálního souboru, nebo zadáním webové adresy vstupní ontologie. V práci jsem se především zaměřil na načtení vstupní ontologie do jádra aplikace a následnou práci s ontologií v jádru aplikace. K tomuto účelu se jako vhodný nástroj jeví OWL API. Tato sada knihoven je Java API a umožňuje nám implementovat kód pro vytváření, manipulaci a serializaci OWL ontologií. OWL API je open source a je dostupná pod licencemi LGPL, nebo licencí Apache. Cílem tedy bylo prozkoumat funkcionalitu těchto knihoven a efektivně ji využít v implementaci výsledného grafického editoru ontologií. Dále jsem se podílel na návrhu a tvorbě samotného GUI aplikace. Pro tento účel jsem využil knihoven Swing. Swing je knihovna uživatelských prvků na platformě Java pro ovládání počítače pomocí grafického rozhraní. Cílem tedy bylo vytvořit uživatelsky přívětivé
a intuitivní prostředí pro práci s výsledným grafickým editorem.

1.3 Struktura práce

V následujícím textu v kapitole 2 nalezneme popis vývojového prostředí, ve kterém vznikala praktická část diplomové práce. Zde nalezneme popis hlavních rysů vývojového prostředí, přičemž hlavní pozornost je věnována oblastem, které byly využity během samotné realizace Frameworku. V kapitole 3 se Vám pokusím nastínit, co je to ontologie. Tento pojem, znám především z filozofie, našel uplatnění v IT a tak celá tato kapitola popisuje ontologie právě z pohledu IT. Kapitola 4 popisuje v současnosti nejpoužívanější jazyk pro popis ontologií. Jazyk je nazván OWL – Ontology Web Language. OWL je značkovacím jazykem vyvinutým organizací W3C (World Wide Web

Consorcium) pro tvorbu ontologií. V této kapitole jsou popsány jednotlivé verze jazyka a následuje popis syntaxe jazyka OWL. Kapitola 5 obsahuje již samotné vypracování Frameworku, který je součástí přiloženého CD i se zdrojovými kódy. V kapitole 6 je shrnut závěr k této diplomové práci. Následuje sekce s použitou literaturou a obsahem přiloženého CD. V poslední části jako příloha k diplomové práci je přiložena ukázka testovací ontologie, která byla využita k otestování správné funkcionality výsledného grafického editoru ontologií.

1.4 Současné editory ontologií

Lze si prakticky představit, že všechny ontologické jazyky můžeme zpracovávat jakýmkoliv textovým editorem. Jako vhodný editor lze také zvolit libovolný XML editor. Ovšem rutinní používání je obtížně představitelné bez specializovaného editoru.

S postupným narůstajícím zájmem o ontologie vznikají různé softwarové nástroje. Tyto nástroje mají za cíl usnadnit tvorbu ontologií, popřípadě jejich editování. V současnosti existuje celá řada podpůrných nástrojů, ale osud většiny z nich bylo, že po představení softwaru na trh byl další vývoj zastaven. Tohle si lze vysvětlit tím, že stále existuje nejednotnost v názorech na reprezentaci znalostních informací.

Následuje přehled v současnosti používaných ontologických editorů:

1.4.1 WebOnto

Jedná se o ontologický editor distribuovaný formou Java appletu, který umožňuje uživatelům prohlížet a editovat znalostní modely na webu. V současné době je WebOnto [*1*] k dispozici jako služba. Byl zhotoven v rámci projekt °u Patman39, HCREMA40 a Enrich41.

1.4.2 OntoEdit

OntoEdit [*2*] je komerční editor s řadou nadstandardních funkcí (např. podpora slučování ontologií, jejich kolaborativní editování apod.). Aplikace je postavena na vrchol interního ontologického modelu. Tento vnitřní model může být serializován pomocí XML, které podporuje práci se soubory. Tento návrhový vzor podporuje reprezentační jazyk pro co nejjednodušší modelování konceptů, vztahů a axiómů. Grafické pohledy na struktury, jež obsahuje podporu ontologického modelování různorodých fází vývoje ontologického cyklu.

1.4.3 Protégé

Velice propracovaný systém, který není bezprostředně spojen s žádným jazykem, avšak je z něj možné exportovat do všech hlavních formátů je Protégé, vyvinutý týmem M. Musena v institutu Stanford Medical Informatics. Protégé [*3*] je bezplatná, open-source platforma, která poskytuje rostoucí uživatelskou komunitu a sadu nástrojů pro konstrukci doménových modelů a knowledge-based aplikací. Ve svém jádru implementuje bohatý soubor knowledgemodeling struktur a akcí, které podporují tvorbu, vizualizaci a manipulaci ontologií v mnoha různých formátech. Protégé může být přizpůsoben k poskytování domain-friendly podpory pro vytváření znalostních modelů a vstupujících dat. Protégé je také možné rozšířit pomocí plug-in architektury a Java-based API pro budování knowledge-based nástrojů a aplikací.

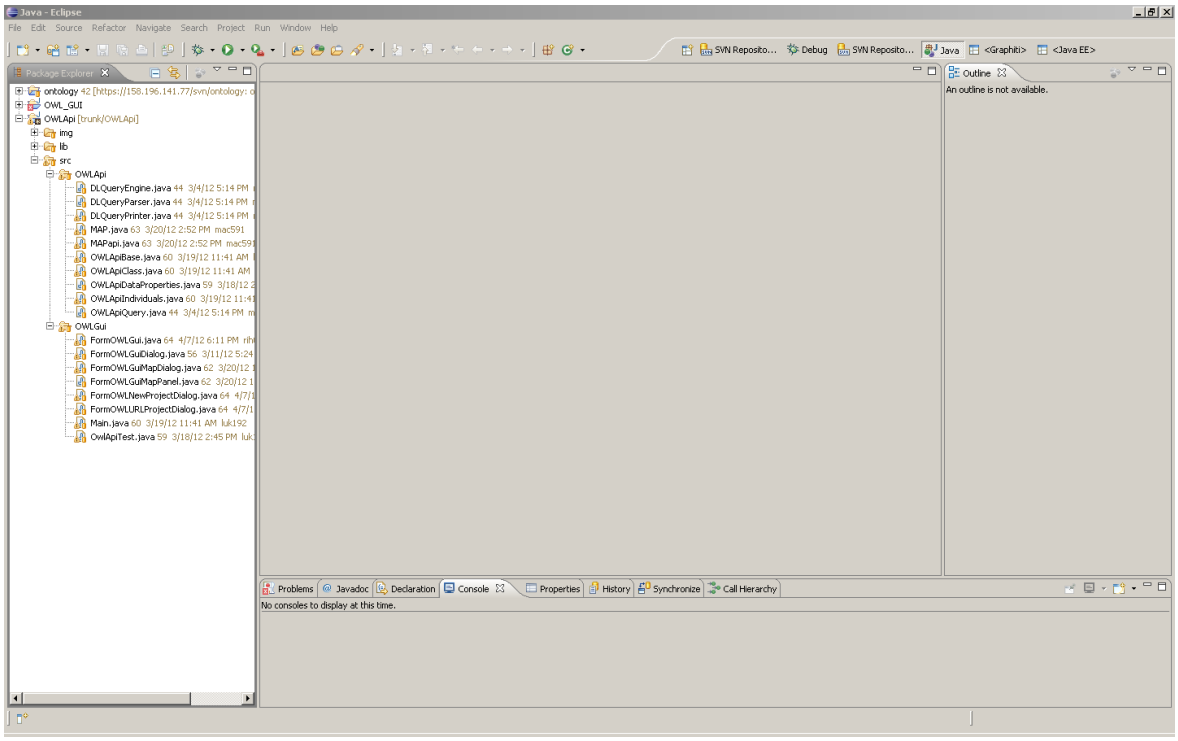
1.4.3.1 Nevýhody Protégé

Poslední verze Protégé nepodporuje některé často využívané pluginy. Bez využití pluginu pro vizualizaci vytvářené ontologie se stává tento nástroj nepřehledný, danou ontologii je obtížné dostatečně zobrazit. Při práci s tímto nástrojem se občas stává, že program hned neaktualizuje změnu v datech. Následuje chybové hlášení. Tento problém lze obejít pomocí zavření a znovu spuštění Protégé, což nemusí každému vyhovovat.

2 Eclipse IDE

2.1 Popis produktu

Vývoj projektu Eclipse byl zahájen na přelomu tisíciletí ve společnosti IBM a v současnosti se jedná o aplikaci šířenou pod licencí EPL. EPL je licence v některých ohledech poněkud odlišná od známé GNU GPL, to však nijak neovlivňuje možnosti použití Eclipse ve funkci IDE. Původně se mělo jednat o integrované vývojové prostředí (IDE), které mělo představovat alternativu, k tehdy již existujícímu prostředí Visual Age, založenému na programovacím jazyku.
Eclipse však je, na rozdíl od zmíněného prostředí Visual Age, postaveno na programovacím jazyku Java [*19*], čímž je zaručena jeho poměrně snadná přenositelnost na různé platformy. Samotné jádro projektu Eclipse je relativně malé, zejména v porovnání s některými „monolitickými“ vývojovými prostředími, ovšem díky koncepci rozšiřujících modulů je možné do Eclipse přidávat další funkce – nové typy editorů, podporu pro další programovací jazyky, ladící nástroje, profilery, návrháře grafického uživatelského rozhraní, napojení na aplikační servery atd.
Společnost IBM navíc v rámci vývoje platformy Eclipse iniciovala i vznik grafického frameworku SWT, který zaručuje, že se Eclipse bude chovat na všech operačních systémech podobně, jako nativní aplikace (což je významný rozdíl oproti frameworku Swing, jenž je standardní součástí JRE i JDK). Mimochodem: framework SWT lze použít i mimo vlastní Eclipse při tvorbě samostatných javovských aplikací a právě díky „nativnímu“ vzhledu je využíván i v mnoha korporátních aplikacích.



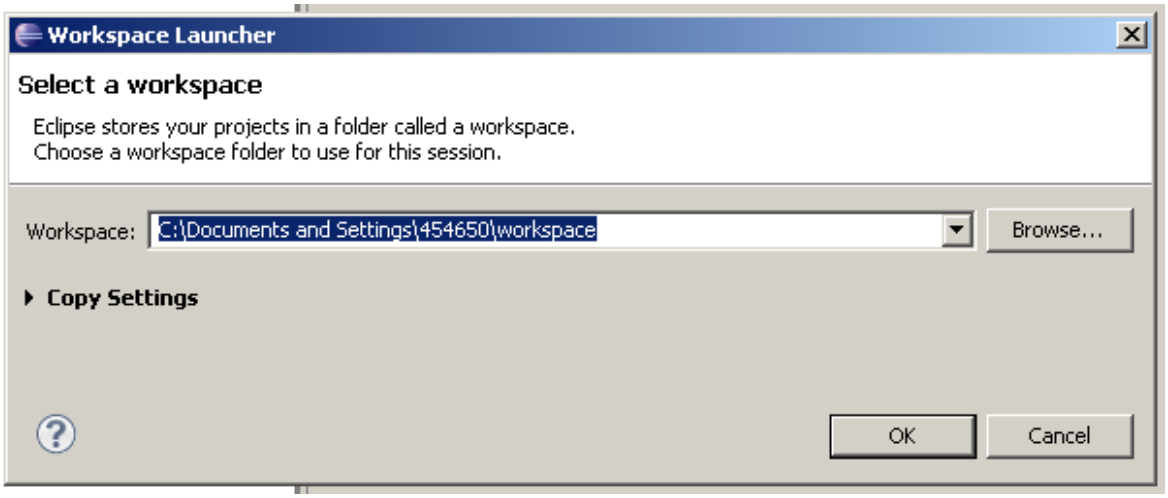
Obrázek 1: Vývojové prostředí Eclipse IDE Indigo Service Release 1

2.2 Popis prostředí

Pracovní plochy (workspaces)

Některé koncepty, které jsou využívané v integrovaném vývojovém prostředí Eclipse, jsou pro tento projekt specifické a mnohdy je v jiných integrovaných vývojových prostředích

nenajdeme. Jedná se především o koncept takzvaných pracovních ploch (*workspaces*), automatické či manuální přepínání mezi perspektivami (*perspectives*) a částečně taktéž o využití pohledů (*views*) a různých interních editorů (*editors*). Nejdříve si řekněme, co se skrývá pod pojmem pracovní plocha. Zjednodušeně řečeno se jedná o adresář (resp. složku), do něhož jsou implicitně ukládány jednotlivé projekty, s nimiž se v Eclipse pracuje.



Obrázek 2: Ukázka pracovní plochy v Eclipse IDE.

Součástí pracovní plochy je taktéž konfigurace vlastního vývojového prostředí i konfigurace jednotlivých zásuvných modulů (*plugins*). S výběrem pracovní plochy se ve skutečnosti setkáte ihned při prvním spuštění Eclipse – prostředí nabídne adresář reprezentující pracovní plochu, který je samozřejmě možné změnit.

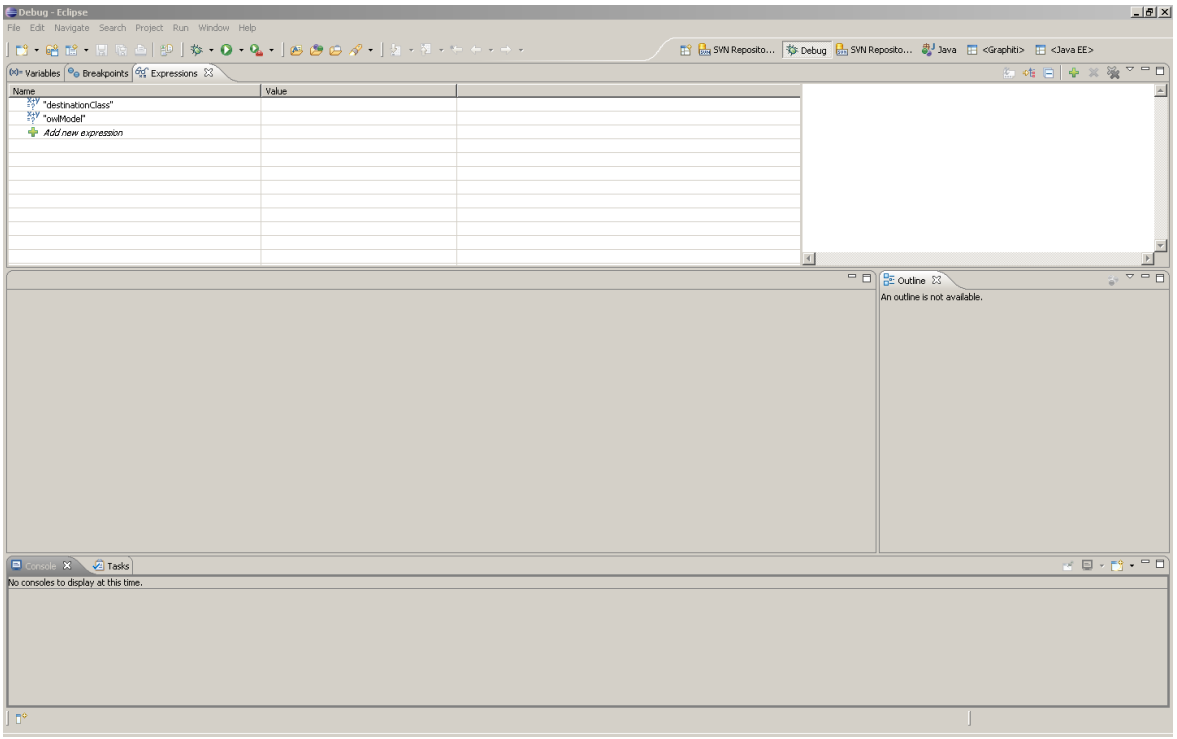
Pracovních ploch může existovat libovolné množství a lze se mezi nimi přepínat s využitím volby File -> Switch Workspace. Při přepnutí na jinou pracovní plochu dojde k restartu celého Eclipse, protože se musí změnit konfigurace celého vývojového prostředí, včetně přídavných modulů.

Perspektivy (perspectives)

Dalším konceptem, s nímž se při práci v Eclipse dříve či později setká každý uživatel, je koncept takzvaných perspektiv (*perspectives*), jejichž princip může být zpočátku pro nové uživatele možná poněkud matoucí, nicméně se jedná o velmi užitečnou technologii. Tvůrci projektu Eclipse si totiž uvědomili, že vývojáři při různých činnostech používají odlišnou konfiguraci pracovní plochy. Například při editaci zdrojového kódu preferují, aby co největší plocha obrazovky byla obsazena vlastním editorem, při ladění aplikace je (kromě zdrojového kódu) taktéž nutné zobrazit obsah proměnných a zásobník volání, dalším režimem činnosti je porovnávání dvou či více verzí zdrojových kódů (zejména při synchronizaci lokální kopie projektu s repositářem) atd.

Aby nebylo s každou změnou režimu činnosti vývojového prostředí (například při přechodu od editace zdrojových kódů k ladění aplikace) nutné složitě měnit způsob rozmístění oken Eclipse na obrazovce, je konfigurace oken i většiny dalších ovládacích prvků ukládána v takzvaných *perspektivách*, přičemž základní nastavení perspektiv (včetně jejich jména) je již připraveno tvůrci jednotlivých přídavných modulů. To však neznamená, že by se jednalo o neměnné nastavení. Ve skutečnosti je tomu právě naopak, protože Eclipse si pamatuje rozmístění oken v každé perspektivě a posléze toto rozmístění při přepnutí perspektiv využije.

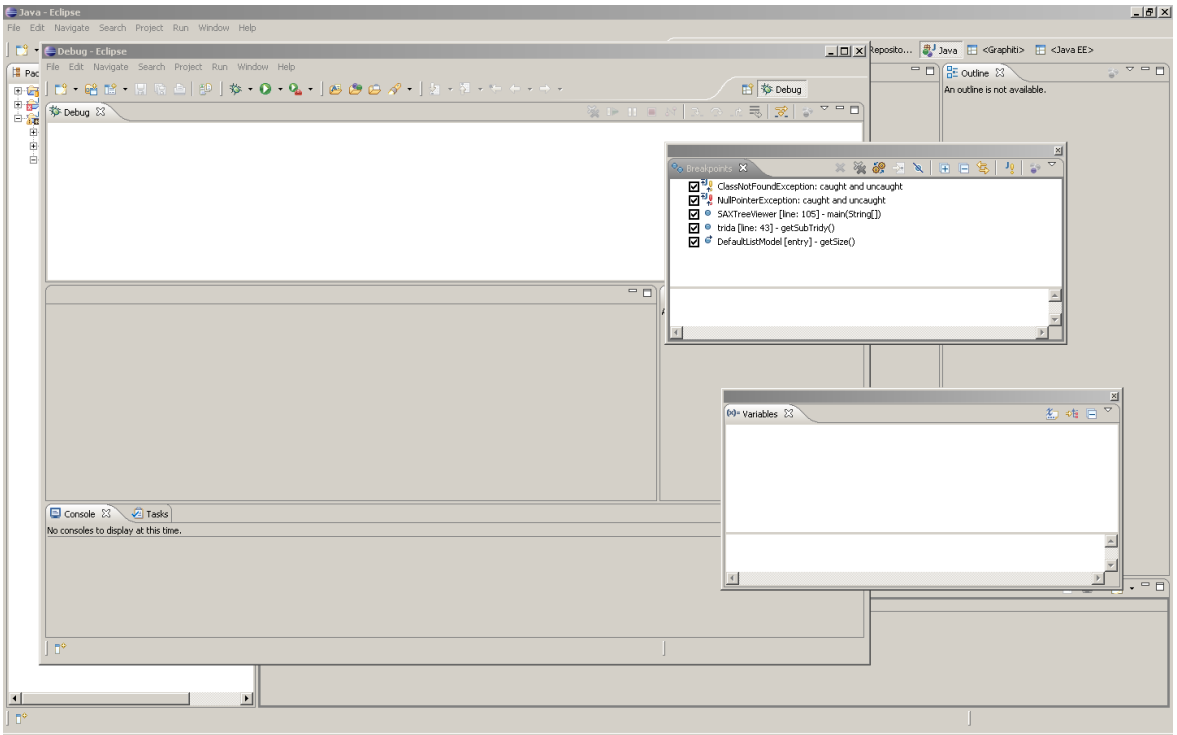
Perspektivu je kdykoli možné přepnout i pomocí ikon zobrazených v pravém horním rohu (velikost plochy pro ikony lze zvětšit myší, což je většinou nutné), popř. lze využít i příkaz Windows → Open Perspective dostupný z hlavního menu Eclipse.



Obrázek 3: Standardní rozložení oken v perspektivě „Java Debug“

Pohledy (views) a editory (editors)

V jednotlivých perspektivách Eclipse jsou zobrazovány prvky grafického uživatelského rozhraní, které se podle své funkce nazývají pohledy (*views*) a editory (*editors*). Pod pojmem *editor* si můžeme představit například běžný textový editor, který je v Eclipse použit pro editaci konfiguračních souborů či souborů s koncovkou .txt. Kromě toho však každý přídavný modul může obsahovat další editory, což je v případě Eclipse JDK především editor zdrojových kódů napsaných v Javě.



Obrázek 4: Pohledy a editory mohou být buď uchyceny v hlavním okně Eclipse IDE, nebo je lze myší „vytáhnout“ z hlavního okna a osamostatnit je.

Pohledem se pak označuje jiný typ prvku grafického uživatelského rozhraní, například prvek zobrazující stromovou strukturu projektu (*Package Explorer*), prvek se strukturou tříd (*Outline*), chybová konzole atd. Většina perspektiv je zpočátku nakonfigurována takovým způsobem, že obsahuje větší plochu určenou pro zobrazení editorů a potom několik okrajových oblastí, do nichž se umisťují pohledy. Jak editory, tak i pohledy lze samozřejmě přemisťovat (nejjednodušší je využít myš) a dokonce je lze vytáhnout z plochy hlavního okna Eclipse do samostatného okna, čehož se často využívá na počítačích s dvojicí monitorů.

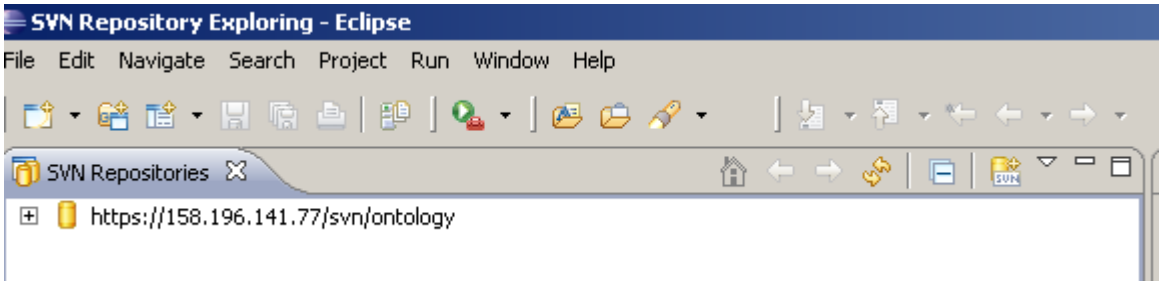
2.3 Použité pluginy

SVN Team Provider Connectors Sources

Do vývojového prosředí Eclipse IDE bylo nutné doinstalovat plugin pro práci s SVN repositářem. K tomuto účelu byl využit plugin od firmy Subversion SVN Team Provider Connectors Sources ve verzi 2.2.2.I20111119-1700. Tento plugin jako nástupce CVS nám dává možnost pracovat s SVN repositářem přímo ve vývojovém prostředí Eclipse IDE.

Mezi klíčové vlastnosti tohoto pluginu mimo jiné patří:

- Realizuje plnou podporu repozitáře, doporučenou Subversion
- Poskytuje pravděpodobnost procházet revize přímo v pohledu umístění úložiště
- Umožňuje zmrazit svn: externals při vytváření značky (tags) a větve (branches)
- Umožňuje automatické vyhledávání projektů Eclipse v úložišti
- Poskytuje podporu SVN operací pro pracovní soubory a soubory projektu
- Subversive plug-in je široce rozšiřitelný a dobře definované API
- Integrace s Mylyn a M2Eclipse plug-iny



Obrázek 5: SVN Repository browser.

Výše uvedené informace v kapitole použité pluginy byly čerpány z [*13*].

3 Ontologie

3.1 Definování ontologie

Ontologie původně definována ve filozofii, má ve své podstatě význam jako filozofická disciplína, která se snaží uspořádat a kategorizovat přirozené bytí. Je považována za univerzální soustavu znalostí popisující objekty, jevy a zákonitosti světa nezávisle na lidském usuzování [*4, 8*]. V IT má ontologie význam jiný. Popisuje to, co „existuje“ a může tedy být reprezentováno v informačním resp. znalostním systému.

Znalostní systém

Můžeme jej interpretovat jako systém pro podporu sdílení a vytváření informací, které jsou důležité například pro rozhodování a chod organizace a to jak v aktuálním dění tak i v budoucím potencionálním rozhodování. Na základě toho pak může organizace zvýšit svou výkonnost díky využití intelektuálního kapitálu firmy uloženého právě v takových systémech. Mezi jeho základní vlastnosti řadíme především tyto:

- Neomezenost – možnost vložit jakoukoli znalost.
- Dobré uspořádání – efektivně nacházet znalosti.
- Souvislost – možnost přistoupit k jakékoliv znalosti v rámci jedné aplikace.

Ontologii lze tedy označit za fundamentální součást a základ znalostního systému organizace, ve kterém jsou uloženy informační objekty a vazby mezi těmito informačními objekty.

Rozšíření znalostních systémů pomoci ontologie

Ontologie mohou být využívány jako další rozšíření znalostních systémů, které vycházejí z konceptuálních schémat definujících význam pojmů, kde je určeno v jakém kontextu se takové pojmy nacházejí. Pod pojmem ontologie si tedy lze představit systém pro ukládání informací, který má jak výše uvedené vlastnosti, tak i některé další:

- Obecnost – možnost aplikace ontologie na základě specifických požadavků.
- Definice pojmů – je to potřeba při velkém počtu pojmů.
- Lze zavést výklad pojmů na encyklopedické úrovni, popřípadě na úrovni výukové či profesní.
- Jazykové znalosti (syntaktické struktury, morfologie) – rozšíření k využití možnosti automatického (strojového) zpracování.

Jedna z uznávaných definic ontologie říká: „*ontologie je explicitní specifikace konceptualizace*“ (T. Gruber 1993) [*7*], tato definice nám říká, že konceptualizace, což je v našem případě definovaný systém objektů modelujících určitou část světa, má být popsána explicitně tedy přímo a autoři by si ji neměli nechávat pouze ve svých hlavách. Definice byla později upravena do této podoby: „*Ontologie je formální, explicitní specifikace sdílené konceptualizace*“ (W. Borst 1997) [10]. V definici se vyskytující explicitní specifikace je nutno popsat formálním způsobem, tedy pomocí daného jazyka disponujícího svou definovanou syntaxí pro využití strojového zpracování. A slovo „sdílená“ nám říká, že ontologie není pouze individuelní záležitostí, ale je spíše shodou určité zájmové skupiny lidí, především pro podporu vzájemné komunikace mezi dvěma „subjekty“.

3.2 Účel ontologií

Jako základní způsoby využití ontologií jsou tradičně jmenovány tyto (viz např. [*5*]):

- usnadnění návrhu znalostně-orientovaných aplikací.

- podpora komunikace (interoperability) mezi počítačovými systémy
- podpora porozumění mezi lidmi (např. experty a znalostními inženýry)

Ve všech třech rolích se ontologie mohou uplatnit v širokém spektru problémových oblastí a úloh.

3.3 Členění ontologií

3.3.1 Podle předmětu formalizace

Ontologie lze rozdělit nejčastěji právě podle předmětu formalizace. Dle různých autorů existuje opět více variant. Mezi základní typy patří:

Generické ontologie

Ontologie se velmi podobají doménovým s tím rozdílem, že mají širší záběr a zachycují obecnější koncepty reality (tedy nejdou příliš do hloubky). Příkladem je možnost reprezentace času, vzájemných pozic objektů nebo meronymických vztahů (vztahy mezi částmi a celkem). Z této kategorie se často ještě vyčleňují ontologie vyšší úrovně (upper-level nebo top-level) usilující o zachycení nejobecnějších pojmů a vztahů, jako základu taxonomie např. doménových ontologií. Příkladem této ontologie je CYC[1] (enCYClopedia). Do této kategorie mohou spadat i takové ontologie, které se snaží zachytit všeobecné znalosti (znalosti typu commonsense) – znalosti běžného rozumu, které používáme v běžné praxi.

Doménové ontologie

Doménové ontologie modelují specifickou část světa (doménu), základem je užší popis významu termínů v dané doméně. Například máme-li slovo „jazyk“, pak existuje mnoho významů, a právě doménové ontologie specifikují význam v dané oblasti jako doména anatomie „část lidského těla“, zatím co v doméně informatiky jako „prostředek pro zápis algoritmů“. Doménové ontologie jako takové jsou nejčastěji používanými ontologiemi.

Úlohové ontologie

Tento typ ontologií se zaměřuje na procesy odvozování, nikoliv na zachycování znalostí o světě. Zaujímají roli modelů řešení problémů, např. pro diagnostiku, konfiguraci nebo plánování.

Aplikační ontologie

Aplikační ontologie je nejspecifičtější. Jedná se o konglomerát modelů převzatých a adaptovaných pro konkrétní aplikaci, zahrnující zpravidla doménovou i úlohovou část (a tím automaticky i generickou část).

3.3.2 Ontologie dle historického vývoje

Terminologické (lexikální) ontologie

Terminologické ontologie jsou charakteristické tím, že kladou důraz hlavně na zachycení taxonomie termínů a vztahů mezi nimi (obecnější a speciálnější pojmy). Příkladem takové ontologie je jistě WordNet.

Informační ontologie

Informační ontologie zastávají roli konceptuální vrstvy pro databázová schémata pro pojmové dotazování.

[1] CYC – projekt založený na generické ontologii, vznikl v roce 1984 s cílem kodifikovat miliony znalostí, které, tvoří lidský rozum.

Znalostní ontologie

Znalostní ontologie navazují na výzkum v oblasti umělé inteligence a jsou v tomto smyslu chápány jako logické teorie a jejich jednotlivé prvky jsou definovány pomocí formálního jazyka. Jsou využívány ve větší míře ve znalostních aplikacích.

3.4 Struktura ontologií

Základní struktura ontologií je ve všech projektech podobná, ale často se liší terminologie. Nejvíce se liší tradiční jazyky jako Ontolingua a nové odlehčené webové jazyky. Dále uvedu prvky tvořící základní strukturu ontologií.

3.4.1 Třídy (koncepty, kategorie)

Třídy označují množiny konkrétních objektů, které jsou specifikovány podmínkami nutnosti i postačitelnosti (příslušnosti individua) se označují jako definované. Ostatní třídy se označují jako primitivní. Třída může obsahovat libovolný počet dalších tříd, může mít libovolný počet atributů. Na množině tříd pak bývá definována taxonomie (hierarchie). Všechny hlavní ontologické jazyky podporují vícenásobnou dědičnost, která se často využívá. Třídy by měly být popsány definicí tak, aby byly jednoznačně vymezeny vůči ostatním. Definice může sloužit i při překladu ontologie do jiného jazyka.

3.4.2 Individuum

Individuum odpovídá konkrétnímu objektu reálného světa. Termín instance je někdy chápán jako ekvivalentní, asociuje ale příslušnost k určité třídě. Konkrétní individuum může být do ontologie vloženo i bez vazby na třídu. Zda je objekt třídou nebo instancí často nezávisí na objektivním stavu světa, ale na úhlu pohledu, kterým se na daný objekt díváme.

3.4.3 Relace, funkce, vlastnosti (sloty)

V ontologiích je také nutnost vymezení vztahů. Lze definovat *relace* označují vztahy mezi třídami nebo objekty. Pomocí logických podmínek a to především v silnějších ontologických jazycích, lze relace definovat. Odlehčené jazyky umožňují relacím přiřadit pouze omezení a to buď globálně, nebo lokálně. V odlehčených jazycích se především u binárních vztahů (relací), tedy vztahů mezi dvěma objekty, definují *vlastnosti* nebo *sloty*. Tyto relace nejsou nějak závislé na určitých třídách.

Zvláštním typem relace jsou takzvané funkce. Poněkud formálněji můžeme funkční sloty definovat jako relace, u nichž je hodnota n-tého argumentu jednoznačně určena předchozími n-1 argumenty. Někdy se funkční slot označuje jako atribut. Platí, že atribut je definován pro všechny instance třídy.

Stejně jako u tříd existují hierarchická uspořádání také u relací. Hierarchie pak spočívá v tom, že argumenty podřízené relace tvoří podmnožinu argumentu nadřízené relace.

3.4.4 Omezení slotů, meta-sloty

Relacím (slotům) je možné přiřazovat vlastnosti, někdy označované jako takzvané *meta-sloty*. Typickým příkladem meta-slotu muže být vyjádření hierarchického vztahu (podřízenosti a nadřízenosti) slotu. Dalšími vlastnostmi slotu jsou mimo jiné *definiční obor* a *obor hodnot*, které jsou vymezeny konkrétními třídami. Takovéto vlastnosti slotů označujeme jako *globální omezení*, protože se vztahují ke slotu bez ohledu na jeho použití.

Často je však třeba omezit hodnoty (např. z konkrétního definičního oboru) slotu aplikovaného na konkrétní třídu. Zejména se jedná o omezení *kardinality* a oboru hodnot slotu. V takovém případě mluvíme o *lokálním omezení* slotů.

3.4.5 Primitivní hodnoty a datové typy

Jak již bylo řečeno, argumenty relací (slotů) mohou být reprezentovány buď pomocí objektů, v takovém případě hovoříme o *objektových slotech*, ale také pomocí *primitivních hodnot*, které žádnému objektu neodpovídají. V tom případě se jedná o takzvané *dato-typové sloty*.

3.4.6 Axiomy, pravidla

V ontologiích se nacházejí výrazy, které explicitně vymezují příslušnost k některým třídám a relacím. Avšak do ontologií lze zařazovat také logické, výrokové formule, které mohou vyjadřovat například ekvivalenci nebo disjunkci tříd, rozklad třídy na podtřídy apod. Tyto formule jsou nazývány *axiomy*, či *pravidly*. V závislosti na dané interpretaci jazyka mohou být axiomy součástmi tříd, nebo mohou vystupovat zcela samostatně.

4 OWL

OWL (Ontology Web Language) je značkovacím jazykem vyvinutým organizací W3C (World Wide Web Consorcium) [*17*] pro tvorbu ontologií využitelných v prostředí sémantického webu. Obsahuje množinu axiomů popisující třídy, vlastnosti a vztahy mezi nimi.

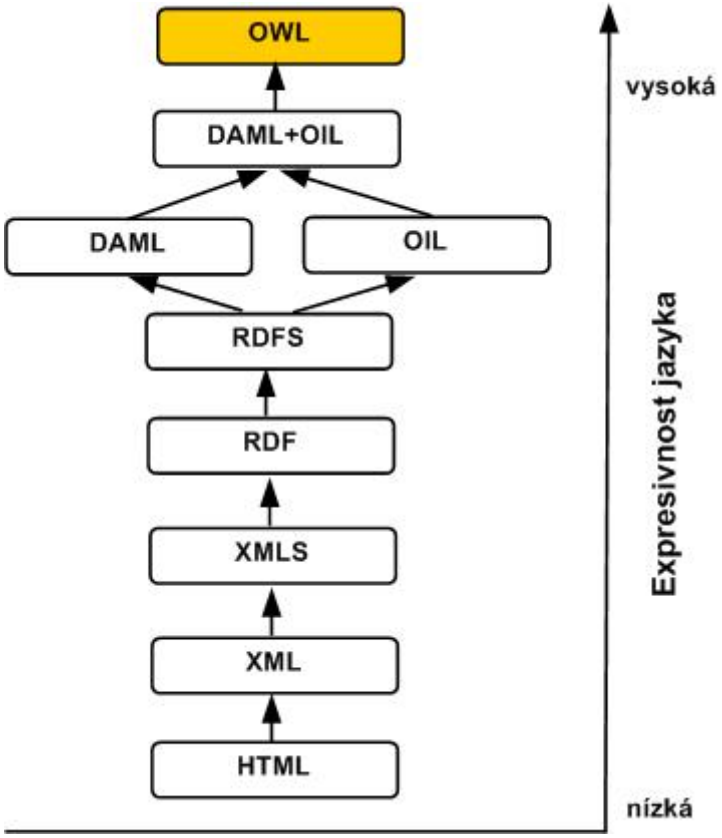
4.1 Ontology Web Language (OWL)

V rámci evropské výzkumné skupiny byl navržen modelovací ontologický jazyk OIL, zatímco americká výzkumná skupina zpracovala jazyk DAML-ONT. Jako sjednocení této snahy vznikl jazyk DAML+OIL, který sloužil jako výchozí bod pro pracovní skupinu W3C Web Ontology pro přípravu ontologického jazyka OWL na rozšíření vyjadřovací síly jazyka RDF a RDFS. Jazyk OWL má být základním ontologickým jazykem sémantického webu.

Dalším cílem mělo být vytvoření jazyka, který by byl ještě expresivnější (tj. měl bohatší sémantiku - slovník pro reprezentaci znalostí) než DAML+OIL a přitom měl být kompatibilní s jazyky XML a RDF. Samozřejmě mělo být možné s jeho pomocí i odvozovat nové skutečnosti z již existujících. Výsledkem byl jazyk OWL.

Pokud se podíváme na sílu expresivity znalostí jazyka OWL mohly bychom ho vidět, v porovnání s předchozími jazyky, na nejvyšší úrovni. Vhodnou ilustrací pro vyjádření této expresivity je následující obrázek:

Obrázek 6: Vyjádření síly expresivity jazyka OWL



Zdroj:[23]

Na OWL je možno pohlížet jako na rozšíření RDF Schéma s tím, že se zachová význam stávajících tříd a vlastností a přidají se nová primitiva do jazyku pro bohatší výrazové schopnosti.

4.1.1 Kompatibilita OWL s RDF a RDFS

Jazyk OWL měl být čistým rozšířením RDFS. Tento jazyk měl využívat vyjadřovací prostředky RDF a RDFS (třídy a vlastnosti) a rozšířit je vlastními výrazy. Při vývoji jazyka bylo ovšem nutné hledat kompromis mezi vyjadřovací silou a efektivním odvozováním. RDFS obsahuje výrazy s velmi vysokou vyjadřovací silou, které by při rozšíření o možnosti jazyka OWL přinášely nepřijatelné výpočetní vlastnosti při odvozování.

Aby jazyk OWL vyhovoval všem kladeným požadavků a zároveň překonal výše uvedená výpočetní omezení, byl tento jazyk navržen ve třech variantách. Nejvyšší varianta nabízí nejvyšší vyjadřovací sílu a následující dvě varianty jsou vždy podmnožinou vyšší varianty.

4.2 Verze OWL

OWL Full

OWL je plnou variantou jazyka OWL, která používá všechny výrazy a konstrukty jazyka OWL a umožňuje je libovolně kombinovat s výrazy RDF a RDFS. Umožňuje také změnit význam výrazů OWL a RDF aplikováním výrazů jazyka navzájem. Výhodou této varianty je plná zpětná sémantická i syntaktická kompatibilita s RDF. Každý RDF dokument je také dokumentem jazyka OWL Full a každý závěr na základě jazyka RDF je také platným závěrem v jazyce OWL. Na druhé straně složitost jazyka vede k nemožnosti úplné výpočetní podpory pro odvozování a vysoké složitosti zpracování jazyka.

OWL DL

Tato verze je kompromisem mezi výpočetní výkonností a vyjadřovací silou. V této variantě není možné aplikovat výrazy jazyka navzájem, což zajišťuje, že jazyk odpovídá standardům deskripční logiky (odtud zkratka DL). Výhoda efektivního zpracovávání jazyka a dobré výpočetní podpory je vyvážena ztrátou plné kompatibility s RDF a RDFS. Zatímco každý platný OWL DL dokument je také platným dokumentem RDF, opačně toto neplatí.

OWL Lite

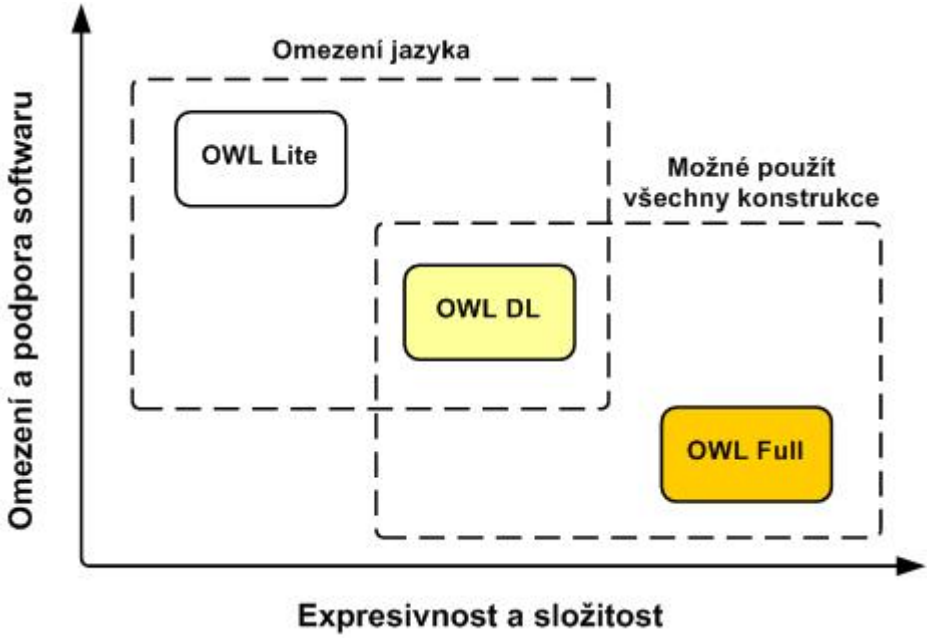
OWL Lite je podmnožinou jazyka OWL DL. Na jazyk byla aplikována další omezení, která snižují vyjadřovací sílu. Jazyk ovšem zjednodušují a přináší snazší a efektivnější zpracovávání. Tento jazyk by měl být snadno přijatelný pro uživatele i výrobce softwarových nástrojů.

Dále také platí pravidla pro všechny tři verze jazyka OWL, která zní takto:

- Každá platná ontologie OWL Lite je platnou ontologií OWL DL.
- Každá platná ontologie OWL DL je platnou ontologií OWL Full.

Na základě toho, jak komplexní otologie budeme tvořit, zvolíme kterou z verzí jazyka využijeme. Chceme-li modelovat komplexnější ontologii, musíme také počítat s tím, že bude potřeba nejen zevrubná znalost plně vybaveného OWL Full, ale také náročnější softwarové vybavení. Obrázek Dialekty OWL jazyka, který je převzat ze zdroje [*6*], vystihuje verze OWL spolu s jejich složitostí a expresivností.

Obrázek 7: Dialekty OWL jazyka



Zdroj:[23]

4.3 Syntaxe OWL

OWL ontologie je ve své podstatě vytvořena pomocí RDF trojic. Definice trojic je tedy stále zachována a postup zápisu je stejný (objekt – predikát – subjekt) a tvoří tedy RDF graf a proto muže být reprezentována kteroukoliv konkrétní syntaxí vhodnou pro RDF. Nejpříhodnější syntaxí, doporučovanou konsorciem W3C, pro reprezentaci RDF trojic je RDF/XML syntaxe. Přípustná je jakákoliv forma této syntaxe.

4.3.1 Hlavička OWL

Hlavička dokumentu ontologie obsahuje některé důležité informace o ontologii. Může obsahovat také komentáře, informace o verzích dokumentu, nebo o vnořených ontologiích. Následující příklad zobrazuje hlavičku OWL dokumentu.

Příklad 1: Hlavička OWL dokumentu.

```
<?xml version="1.0"?>
<!DOCTYPE Ontology [
    <!ENTITY xsd "http://www.w3.org/2001/XMLSchema#" >
    <!ENTITY xml "http://www.w3.org/XML/1998/namespace" >
    <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema#" >
    <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns#" >
]>
```

Dále může následovat OWL záhlaví s různými informacemi jako jsou komentáře, čísla verzí a vložení dalších ontologií. Veškeré tyto údaje jsou zahrnuty pod elementem *owl:Ontology*. V následujícím příkladu jsou tyto vlastnosti předvedeny.

Příklad 2: Další součást Hlavičky OWL dokumentu.

```
<owl:Ontology rdf:about="">
      <rdfs:comment>Firemní ontologie zaměřená na ... </rdfs:comment>
```

```
       <owl:priorVersion
         rdf:resource="http://www.onto.com/old"/>
       <owl:imports
         rdf:resource="http://www.ontologie.org/zamestnanci"/>
      <rdfs:label>Firemní ontologie</rdfs:label>
      </owl:Ontology>
```

4.3.2 Třídy

Chápání pojmu Třída v OWL není odlišná od objektově orientovaného principu. Jedná se o soubor jedinců se stejnými vlastnostmi. Jsou organizovány do hierarchie - taxonomie tj. nadtříd a podtříd (superclasses a subclasses). Třída je základním hierarchickým prvkem ontologie. Třídy jsou v OWL definovány pomocí *owl:Class* elementu. Jednou z předdefinovaných je *owl:Thing*. Jedná se o tzv. systémovou nejobecnější třídu, do které spadají všechny ostatní. Je to jakýsi společný předek všech tříd, které vytvoříme.

Příklad 3: Definice třídy v OWL

```
<owl:Class rdf:ID="Zamestnanec">
  <owl:subClassOf rdf:resource="#programovyAnalytik">
</owl:Class>
```

Chceme-li pomocí OWL definovat například disjunktnost (nesouvislost) tříd, pomocí kterých určujeme se kterými třídami daná popisovaná třída nesouvisí, využijeme pro tento účel popis *owl:disjointWith.*

Příklad 4: Definice třídy v OWL

```
<owl:Class rdf:about="#externiProgramator">
    <owl:disjointWith rdf:resource="#programator"/>
    <owl:disjointWith rdf:resource="#programovyAnalitik"/>
   <owl:disjointWith rdf:resource="#vedouciProgramator"/>
</owl:Class>
```

Definice ekvivalence uvnitř třídy, je další z možností jak vyjádřit třídu. Lze ji provést pomocí vestavěné vlastnosti *owl:equivalentClass.* Smysl této definice je takový, že sady prvků dvou v sobě zahrnutých tříd jsou stejné.

Příklad 5: Definice ekvivalence tříd v OWL

```
<owl:Class rdf:about="#US_President">
  <equivalentClass
   rdf:resource="#PrincipalResidentOfWhiteHouse"/>
</owl:Class>
```

Pomocí seznamu (výčtu) prvků uvnitř třídy lze také v OWL definovat třídu. K takové konstrukci se užívá vlastnost *owl:oneOf.*

Příklad 6: Definice třídy s výčtem prvků v OWL.

```
<owl:Class>
<owl:oneOf rdf:parseType="Collection">
  <owl:Thing rdf:about="#SecurityProgram"/>
```

```
  <owl:Thing rdf:about="#DesignProgram"/>
  <owl:Thing rdf:about="#TradingProgram"/>
  <owl:Thing rdf:about="#AudioProgram"/>
  <owl:Thing rdf:about="#VideoProgram"/>
</owl:oneOf>
</owl:Class>
```

4.3.3 Vlastnosti

Vlastnosti jsou jistým druhem binárních relací, díky kterým lze ke třídám a jejich prvkům přidávat tvrzení a spojení mezi jednotlivými objekty a datovými typy. Obecně má definice vlastnosti následující formát.

Příklad 7: Obecná definice vlastnosti.

```
<owl:ObjectProperty rdf:ID=”název vlastnosti”>
Další podmínky a typy vlastnosti
</owl:ObjectProperty>
```

Vlastnost, která spojuje dva objekty, bývá označována jako objektová vlastnost a lze ji definovat elementem *owl:ObjectProperty.*

Příklad 8: Příklad objektové vlastnosti.

```
<owl:ObjectProperty rdf:ID="jeVyvijen">
   <rdfs:domain rdf:resource="#program"/>
   <rdfs:range rdf:resource="#zamestnanec"/>
   <rdfs:subPropertyOf rdf:resource="#obstarava"/>
</owl:ObjectProperty>
```

Vlastnost, která spojuje určitý objekt s nějakým datovým typem, označujeme jako dato-typová vlastnost a definujeme ji pomocí elementu *owl:DatatypeProperty*.

Příklad 9: Příklad dato-typové vlastnosti.

```
<owl:DatatypeProperty rdf:ID="telefon">
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema
  #Integer"/>
</owl:DatatypeProperty>
```

4.3.4 Omezení v OWL

Omezení se provádí pomocí tříd omezení, která mají definována dané restrikce pro své vlastnosti. Tyto restrikce přímo určují, která individua ze třídy přímo zahrnout nebo vyjmout. Obecný formát definice třídy omezením vlastností vypadá následovně.

Příklad 10: Definice omezení třídy.

```
<owl:Restriction>
  <owl:onProperty rdf:resource="#hasSugar"/>
  <owl:hasValue rdf:resource="#Sweet">
</owl:Restriction>
```

Globální omezení

Pomocí definičního oboru a oboru hodnot lze specifikovat, mezi kterými třídami, resp. jedinci může vlastnost existovat. Dá se říci, že jejich specifikací určitým způsobem samotnou vlastnost omezujeme, říkáme jí, kde je její místo, mezi čím může působit. Definiční obor a obor hodnot jsou tzv. globální omezení vlastnosti.

Lokální omezení vlastnosti

Kromě globálních omezení vlastnosti existují ještě lokální omezení vlastnosti, která se vyskytují tam, kde vlastnost vystihuje sémantiku (význam) třídy. K těmto omezením patří:

- kvantitativní omezení vlastnosti
  - existenciální (existenční) omezení vlastnosti
  - univerzální omezení vlastnosti
- kardinalitní omezení vlastnosti
  - minimum (>=)
  - maximum (<=)
  - rovnost (=)
- hasValue omezení

Příklad 11: Příklad omezení vlastností.

```
<owl:Restriction>
  <owl:onProperty rdf:resource="#madeFromGrape"/>
  <owl:maxCardinality
    rdf:datatype="http://www.w3.org/2001/XMLSchema#nonNeg
    ativeInteger">1
  </owl:maxCardinality>
</owl:Restriction>
```

Univerzální omezení

Jedná se o typ omezení, které nese označení only, no values except, onlyValuesFrom. Na rozdíl od existenciálního omezení vůbec negarantuje existenci vztahu s nějakou třídou. Třída, kterou vymezujeme může mít vztah s nějakou třídou, ale nemusí. Pokud už nějaký vztah má, tak se jedná jen o tento vztah s danou třídou a žádná jiná třída nepřipadá v úvahu.

Kardinalitní omezení

Pomocí tohoto omezení můžeme vyčíslit počet vztahů, kterých se jedinec, náležející určité třídě, má účastnit. Je jednodušší na pochopení než existenciální omezení. Omezení lze rozdělit na:

- minimální kardinalitní omezení (v OWL Lite: 0 nebo 1);
- maximální kardinalitní omezení (v OWL Lite: 0 nebo 1);
- omezení s rovností (v OWL Lite: 0 nebo 1).

Příklad 12: Příklad omezení kardinality.

```
<owl:Class rdf:about="#program">
 <rdfs:subClassOf>
  <owl:Restriction>
   <owl:onProperty rdf:resource="#jeVyvijen"/>
   <owl:minCardinality
      rdf:datatype="&xsd;nonNegativeInteger">
```

```
    </owl:minCardinality>
   </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

5 Vlastní řešení Frameworku

V této části se dostáváme k popisu implementace Frameworku. Vize vytvořeného Frameworku je zobrazit si ontologii tak, jak si uživatel předem nadefinuje jednotlivé zobrazené elementy ontologie. Tedy uživatel si sám zvolí grafické prvky, jak se mu načtená ontologie zobrazí v grafickém editoru ontologií.

Kapitolu vlastní řešení Frameworku jsem rozdělil na dvě podkapitoly. První z nich popisuje problematiku samotného načtení ontologie do jádra aplikace. K tomu, aby uživatel mohl pracovat s ontologií a následně si definovat jak se mu zobrazí jednotlivé individua, třídy, či jak zobrazí jednotlivé vazby mezi nimi, je nejprve nutné danou ontologii načíst do jádra aplikace. Důležitým aspektem je ontologii načíst tak, aby nedošlo ke ztrátě informace. Všechny atributy musí být objektově přístupné, aby v další fázi bylo možné s těmito prvky pracovat.

V druhé části kapitoly vlastní řešení Frameworku je podkapitola popisující návrh GUI [*20*] pro zobrazení načtené ontologie. Důraz je zde kladen na přehledné zobrazení jednotlivých atributů ontologie. Pro implementaci jsem zvolil sadu knihoven Swing. Tato knihovna je vhodná pro implementaci grafického rozhraní na platformě Java. Pomocí Swingu je možno vytvářet okna, dialogy, tlačítka, rámečky, rozbalovací seznamy, a další grafické komponenty.

5.1 OWL API

Sada knihoven OWL API [*21*] je v Javě implementované API a mezi klíčové vlastnosti lze zařadit vytváření, editaci a serializaci OWL ontologií. OWL API ve verzi 3 je orientována na syntaxi OWL ve verzi 2. Jak bylo uvedeno v kapitole 1, sada knihoven je spravována pod licenci LGPL a Apache licencí.

OWL API umožňuje práci s ontologiemi, které získá z různých datových formátů. Obsahuje tyto komponenty:

- API pro práci s ontologií v syntaxi OWL 2
- Čtení a zápis RDF/XML
- Čtení a zápis OWL/XML
- Čtení a zápis Turtle
- Čtení a zápis OWL Functional Syntax
- Čtení OBO Flat
- Rozhraní pro prácí s FaCT++, HermiT, Pellet a Racer

V OWL API je také možnost do vytvořené ontologie přidávat vlastnosti, třídy, objekty a editovat je či mazat. Mezi pokročilejší funkce rozhraní patří například sloučení více ontologií do jedné.

OWL API je použit v mnoha známých sémantických aplikacích jako například v OWL editorech Protégé 4, Pallet, SWOOP či OntoTrack.

Výše uvedené informace v kapitole OWL API byly čerpány z [*12*].

5.1.1 Načtení ontologie do jádra aplikace

Po odzkoušení několika knihoven pro práci s ontologií, se jako nejvhodnější nástroj pro práci se vstupní ontologií jeví sada knihoven OWL API. Tato sada knihoven je spravována a dále vyvíjena veřejnou výzkumnou univerzitou The University of Manchester [*11*]. S nabytími vědomostmi o ontologiích viz kapitola 3 a se znalosti syntaxe OWL jazyka viz. kapitola 4, se právě tato sada knihoven jeví jako vhodný nástroj pro řešení námi definované problematiky. V práci je použita sada knihoven OWL API ve verzi 3.

Budeme-li si chtít načíst vstupní ontologii využijeme k tomuto účelu rozhraní *OWL Ontology Manager*. Pomocí *OWL Ontology Manager* můžeme spravovat soubor vstupních ontologií. Toto rozhraní bude pro nás hlavním bodem pro práci s ontologií. Mimo jiné, je možné využití tento interface pro vytváření ontologie, také lze spravovat mapovaní mezi načtenou ontologií a vstupním dokumentem s ontologií.

Následuje praktická ukázka z třídy OWLApiBase z balíčku OWLApi, jak si načíst vstupní ontologii pomocí výše popsaného *OWL Ontology Manager*. Jako parametr *Document_IRI* je potřeba předat IRI OWL ontologie. Je možné načítat ontologie jak z URL, tak i z lokálního úložiště.

Příklad 13: Načtení OWL ontologie.

```
public class OWLApiBase {
private static String DOCUMENT_IRI =
"http://owl.man.ac.uk/2005/07/sssw/people";


public static String getDOCUMENT_IRI()
      {
      return DOCUMENT_IRI;
      }


public static void setDOCUMENT_IRI(String dOCUMENT_IRI)
      {
      DOCUMENT_IRI = dOCUMENT_IRI;
      }
private static OWLOntologyManager manager;
private static IRI docIRI;
public static OWLOntology ont;
public OWLApiBase(String OWLFile)
      {
      if (OWLFile != "")
      {
      setDOCUMENT_IRI(OWLFile);
      }
      try {
            manager = OWLManager.createOWLOntologyManager();
            docIRI = IRI.create(DOCUMENT_IRI);
```

```
            ont = manager.loadOntologyFromOntologyDocument(docIRI);
            System.out.println("Načteno: " + ont.getOntologyID());
            Factory = manager.getOWLDataFactory();
          } catch (OWLOntologyCreationException e)
                {
                e.printStackTrace();
                }
      }
}
```

5.1.2 Práce s OWL ontologií

OWL ontologie je soubor axiomů, které poskytují explicitní logické tvrzení o třech typech věcí – tříd, individuí a vlastností. Pro samotnou práci s OWL ontologií je nutné využít *Reasoner* rozhraní. Toto rozhraní je klíčovým prvkem pro práci s OWL ontologií. V celém Frameworku všechny dotazy nad načtenou ontologii OWL jsou provedeny právě pomocí *Reasoner*. Chceme-li obdržet správné výsledky dotazu, je nutné dotazovat se přes námi vybrané rozhraní. V implementaci je použit HermiT, volně k dispozici z [*14*].

Následuje praktická ukázka z třídy OWLApiBase z balíčku OWLApi, jak zpracovat vstupní ontologii pomocí *Reasoner*. Jako parametr, je nutné předat načtenou ontologii, jak bylo uvedeno v kapitole 5.1.1. V našem příkladu je to proměnná *ont*.

Příklad 14: Použití Reasoner.

```
private static OWLReasonerFactory reasonerFactory;
private static ConsoleProgressMonitor progressMonitor;
private static OWLReasonerConfiguration config;
public static OWLReasoner reasoner;

      { //tělo metody OWLApiBase
      progressMonitor = new ConsoleProgressMonitor();
      config = new SimpleConfiguration(progressMonitor);
      reasoner = reasonerFactory.createReasoner(ont, config);
      reasoner.precomputeInferences();
      }
```

5.1.2.1 Načtení tříd

Třída definuje skupinu individuí, kteří patří do jedné třídy, protože sdílejí některé vlastnosti. Například Petr a Pavel jsou individua třídy *Osoba*. K dispozici je i nejobecnější třída s názvem *Thing*. Do této třídy patří všechna individua a je super třída všech tříd v OWL. Naopak existuje třída nazvaná *Nothing*, která nemá žádné instance, ani podtřídy všech tříd OWL.

Následuje praktická ukázka z třídy OWLApiBase z balíčku OWLApi, jak načíst OWL třídy z OWL ontologie pomocí metody *getOWLApiClass*. Jako parametr se zde předává parametr *IRI iri,* pomocí něhož identifikujeme, které třídy se mají načíst. Návratová hodnota je v našem případě atribut *retClass*, který v případě existence OWL třídy nám vrátí OWL třídu, jinak je návratový typ *Null*.

Příklad 15: Načtení OWL tříd pomocí getOWLApiClass metody.

```
public static OWLApiClass getOWLApiClass(IRI iri)
    {
      OWLApiClass retClass = null;
      for (int i = 0; i < owlApiClasses.size(); i++)
      {
         IRI myIRI = owlApiClasses.get(i).getInstance().getIRI();
         if (myIRI.toString().compareToIgnoreCase(iri.toString()) == 0)
            {
                  retClass = owlApiClasses.get(i);
                  break;
            }
      }
      return retClass;
    }
```

5.1.2.2 Načtení vlastností

Vlastnosti můžeme využít k získaní informací ohledně vztahů mezi individui, nebo mezi individui a datových hodnot. Jako příklad *vlastnosti* můžeme uvést vztah *máDítě*, tuto vlastnost můžeme použít pro instance třídy Osoba.

Následuje praktická ukázka z třídy OWLApiBase z balíčku OWLApi, jak načíst OWL vlastnosti z OWL ontologie pomocí metody *getOWLApiProperie*. Jako parametr se zde předává parametr *IRI iri,* pomocí něhož identifikujeme, který datový typ se má načíst. Návratová hodnota je v našem případě atribut *retPropertie*, který v případě existence OWL vlastnosti nám vrátí OWL vlastnost, v opačném případě je návratový typ *Null*.

Příklad 16: Načtení OWL vlastností pomocí metody getOWLApiProperie.

```
public static OWLApiDataProperties getOWLApiProperie(IRI iri)
    {
      OWLApiDataProperties retPropertie = null;
      for (int i = 0; i < owlApiDataProperties.size(); i++)
      {
          IRI myIRI = owlApiDataProperties.get(i).getInstance().getIRI();
          if (myIRI.toString().compareToIgnoreCase(iri.toString()) == 0)
             {
                retPropertie = owlApiDataProperties.get(i);
                break;
             }
       }
       return retPropertie;
    }
```

5.1.2.3 Načtení individuí

Individua jsou instance tříd a pomocí vlastností, můžeme jedno individuum přiřadit k druhému individuu. Jako příklad můžeme individuum *Pavel* popsat jako instanci třídy *Osoba* a vlastnost *jeZaměstnán* můžeme přiřadit k individui *Pavel*.

Následuje praktická ukázka z třídy OWLApiBase z balíčku OWLApi, jak načíst OWL individua z OWL ontologie pomocí metody *getOWLApiIndividual*. Jako parametr se zde předává parametr *IRI iri,* pomocí něhož identifikujeme, které individua se mají načíst. Návratová hodnota je v našem případě atribut *retIndividual*, který v případě existence OWL individua nám vrátí OWL individua, v opačném případě je návratový typ *Null*.

Příklad 17: Načtení OWL individuí pomocí metody getOWLApiIndividual.

```
public static OWLApiIndividuals getOWLApiIndividual(IRI iri)
    {
      OWLApiIndividuals retIndividual = null;
      if (owlApiIndividuals != null)
      {
          for (int i = 0; i < owlApiIndividuals.size(); i++)
          {
          IRI myIRI = owlApiIndividuals.get(i).getInstance().getIRI();
          if (myIRI.toString().compareToIgnoreCase(iri.toString()) == 0)
            {
               retIndividual = owlApiIndividuals.get(i);
               break;
             }
          }
      }
      return retIndividual;
    }
```

5.1.2.4 Naplněni listu třídami

Pomocí metody *getOWLApiClass* jsme získali všechny třídy v OWL ontologii. Seznam těchto tříd si uložíme do *ArrayList owlApiClasses* metodou *fillOWLApiClasses.*

Následuje praktická ukázka z třídy OWLApiBase z balíčku OWLApi, která uloží OWL třídy z OWL ontologie pomocí metody *fillOWLApiClasses* do *ArrayList owlApiClasses*. Metoda je typu *Void*, nemá tedy návratovou hodnotu.

Příklad 18: Naplnění ArrayList třídami OWL.

```
public static void fillOWLApiClasses()
      {
            owlApiClasses = new ArrayList<OWLApiClass>();
            Node<OWLClass> topNode = reasoner.getTopClassNode();
            OWLClass parent = null;
        GetOWLNodes(topNode, parent, null);
      }
```

5.1.2.5 Naplnění OWL tříd do stromu

Načtené OWL třídy budeme chtít zobrazit ve stromové struktuře. K tomuto účelu se hodí využít komponenta *Tree*. Pomocí konstruktoru *DefaultMutableTreeNode* naplníme stromovou strukturu OWL tříd. Využijeme k tomu metodu *GetOWLNode*.

Následuje praktická ukázka z třídy OWLApiBase z balíčku OWLApi, která uloží OWL třídy z OWL ontologie pomocí metody *GetOWLNode* do proměnné *jNode*, která je instancí *DefaulMutableTreeNode*.

Příklad 19: Naplnění OWL tříd do stromu pro GUI aplikace.

```
private static DefaultMutableTreeNode GetOWLNode(Node<OWLClass> node,
OWLClass parent, DefaultMutableTreeNode jParent)
    {
            DefaultMutableTreeNode jNode = null;
            OWLApiClass objectTrida;
            for(Iterator<OWLClass> it = node.getEntities().iterator();
it.hasNext(); )
         {
            OWLClass cls = it.next();
            objectTrida = new OWLApiClass(renderer.render(cls), cls,
parent);
            owlApiClasses.add(objectTrida);
            for (OWLApiIndividuals in:  objectTrida.getIndividua())
            {
                  addOWLApiIndividualsToList(in);
            }
            jNode = new DefaultMutableTreeNode(objectTrida);
            if (jParent == null)
            {
                  DefaultMutableTreeNode root = jNode;
                  treeOWLClasses = new JTree(root);
            }
            else
            {
                  jParent.add(jNode);
            }
         }
            return jNode;
    }
```

5.1.2.6 Uložení tříd do ArrayListu

Pomocí metody *GetOWLNodes* si projdeme celou OWL ontologii rekurzivně a následně ji uložíme do *ArrayListu* tříd.

Následuje praktická ukázka z třídy OWLApiBase z balíčku OWLApi, která uloží OWL třídy z OWL ontologie pomocí metody *GetOWLNodes* do *ArrayList OWLClass*. Metoda je typu *Void*, nemá tedy návratovou hodnotu.

```
private static void GetOWLNodes(Node<OWLClass> node, OWLClass parent,
DefaultMutableTreeNode jTreeParent)
    {
        if(node.isBottomNode())
        {
            return;
        }
        DefaultMutableTreeNode jNode = GetOWLNode(node, parent,
jTreeParent);
        OWLClass cl = node.getRepresentativeElement();
        for(Node<OWLClass> child :
reasoner.getSubClasses(node.getRepresentativeElement(), true))
        {
            GetOWLNodes(child, cl, jNode);
        }
    }
```

5.1.2.7 Naplnění seznamu vlastností

Pomocí metody *getOWLApiProperie* jsme získali všechny vlastnosti v OWL ontologii. Seznam těchto vlastností si teď uložíme do *ArrayList owlApiDataProperties* metodou *fillOWLApiDataProperties*.

Následuje praktická ukázka z třídy OWLApiBase z balíčku OWLApi, která uloží OWL vlastnosti z OWL ontologie pomocí metody *fillOWLApiDataProperties* do *ArrayList owlApiDataProperties*. Metoda je typu *Void*, nemá tedy návratovou hodnotu.

Příklad 20: Naplnění seznamu DataProperties.

```
public static void fillOWLApiDataProperties()
      {
            owlApiDataProperties = new ArrayList<OWLApiDataProperties>();


            OWLApiDataProperties dpRoot = new OWLApiDataProperties();
            dpRoot.setNazev("Data propertie");
            DefaultMutableTreeNode root = new
DefaultMutableTreeNode(dpRoot);
            treeOWLDataProperties = new JTree(root);
            for (OWLObjectProperty op :
OWLApiBase.ont.getObjectPropertiesInSignature(true))
            {
                  OWLApiDataProperties dp = new OWLApiDataProperties();
                  dp.setInstance(op);
                  dp.setNazev(OWLApiBase.renderer.render(op));
```

```
                owlApiDataProperties.add(dp);
                DefaultMutableTreeNode jNode = new
DefaultMutableTreeNode(dp);
                root.add(jNode);
            }
        }
```

5.1.2.8 Přidání individua do listu

Pomocí metody *addOWLApiIndividualsToList* budeme přidávat individua do *ArrayList owlApiIndividuals* v případě, že dané individuum není již v tomto *ArrayList owl ApiIndividuals* obsaženo.

Následuje praktická ukázka z třídy OWLApiBase z balíčku OWLApi, která přidá OWL individua z OWL ontologie pomocí metody *addOWLApiIndividualsToList* do *ArrayList owlApiIndividauls,* pokud dané individuum není již prvkem *owlApiIndividuals*. Metoda je typu *Void*, nemá tedy návratovou hodnotu.

Příklad 21: Naplnění listu pomocí metody addOWLApiIndividualsToList.

```
public static void addOWLApiIndividualsToList(OWLApiIndividuals
individual)
  {
            if (owlApiIndividuals == null)
            {
                  owlApiIndividuals = new ArrayList<OWLApiIndividuals>();
            }

            boolean checkIndividual = false;
            for (int i = 0; i < owlApiIndividuals.size(); i++)
      {
            IRI myIRI = owlApiIndividuals.get(i).getInstance().getIRI();
            if
(myIRI.toString().compareToIgnoreCase(individual.getInstance().getIRI().t
oString()) == 0)
                  {
                  checkIndividual = true;
                  break;
                  }
      }
      if (!checkIndividual)
      {
            owlApiIndividuals.add(individual);
      }
  }
```

5.1.2.9 Naplnění stromu individui

Pomocí metody *getOWLApiIndividual* jsme získali všechny individua v OWL ontologii. Seznam těchto individuí si teď uložíme do *ArrayList owlApiIndividuals* metodou *fillOWLApiIndividuals.*

Následuje praktická ukázka z třídy OWLApiBase z balíčku OWLApi, která uloží OWL individua z OWL ontologie pomocí metody *fillOWLApiIndividuals* do *ArrayList owlApiIndividuals*. Metoda je typu *Void*, nemá tedy návratovou hodnotu.

Příklad 22: Naplnění seznamu individuí.

```
public static void fillOWLApiIndividuals()
      {
            if (owlApiIndividuals == null)
            {
                  owlApiIndividuals = new ArrayList<OWLApiIndividuals>();
            }
            OWLApiIndividuals iRoot = new OWLApiIndividuals();
            iRoot.setNazev("Individuals");
            DefaultMutableTreeNode root = new
DefaultMutableTreeNode(iRoot);
            treeOWLIndividuals = new JTree(root);
            for (OWLApiIndividuals in : owlApiIndividuals)
            {
              DefaultMutableTreeNode jNode = new
DefaultMutableTreeNode(in);
              root.add(jNode);
            }
      }
```

5.1.3 Práce s třídami

Pro potřeby našeho Frameworku pro grafický editor ontologií není postačující si načíst axiomy ze vstupní OWL ontologie, jak bylo předvedeno v předchozí kapitole 5.1.2. Mluvíme-li o OWL třídách, je nutné hlouběji a podrobně zkoumat jednotlivé vztahy mezi třídami. Zaměříme se tedy na vztahy podtříd, disjunktivních tříd, ekvivalentních tříd, atd. Definice tříd, podtříd a dalších matematických výrazů, je dostupné z [*18*]. V každé následující podkapitole si nejprve přesně nadefinujeme danou vlastnost a pak ukážeme v kódu implementaci pomocí dané metody.

5.1.3.1 Podtřída

Uveďme si nejdřív slovně význam podtřídy v OWL ontologii. Je-li třída C1 definována jako podtřída třídy C2, pak soubor individuí ve třídě C1 by měli být podmnožinou množiny jedinců ve třídě C2.

Tento slovní popis lze zachytit takto:

Axiom	Podmínka
SubClassOf (CE_1 CE_2)	$(CE_1)^C \subseteq (CE_2)^C$

kde *SubClassOf* označuje podtřídu. Symbol *CE* označuje *Class Expression*, tedy danou třídu a symbol *C* označuje *Class,* třídu. Symbolem ⊆ značíme podmnožinu.

Následuje praktická ukázka z třídy OWLApiClass z balíčku OWLApi, kde pomocí metody *getSubTridy* obdržíme všechny podtřídy námi vybrané třídy.

Příklad 23: Naplněni listu podtřídami třídy.

```
public ArrayList<OWLApiClass> getSubTridy() {

      ArrayList<OWLApiClass> subClassArray = new
ArrayList<OWLApiClass>();

      OWLClass myClass =
OWLApiBase.factory.getOWLClass(this.instance.getIRI());

      for (OWLClass subClass : OWLApiBase.reasoner.getSubClasses(myClass,
true).getFlattened())

      {

                  OWLApiClass myApiClass =
OWLApiBase.getOWLApiClass(subClass.getIRI());

                  if (myApiClass == null)

                  {

                        myApiClass = new
OWLApiClass(OWLApiBase.renderer.render(subClass), subClass, null);

                  }

            subClassArray.add(myApiClass);

      }

      return subClassArray;

}
```

5.1.3.2 Disjunktivní třída

Chceme-li výpis disjunktivních tříd k námi vybrané třídě, pak platí, že žádné individuum námi zvolené třídy, nemůže být zároveň instancí jiné třídy.

Tento slovní popis lze zachytit takto:

Axiom	Podmínka
DisjointClasses ($CE_1 \dots CE_n$)	$(CE_j)^C \cap (CE_k)^C = \emptyset$ pro každé $1 \le j \le n$ a pro každé $1 \le k \le n$, kde $j \ne k$

kde *DisjointClasses* označuje disjunktivní třídu. Symbol *CE* označuje *Class Expression*, tedy danou třídu a symbol *C* označuje *Class,* třídu. Symbolem ∅ označujeme prázdnou množinu. Symbolem ∩ označujeme průnik množin.

Následuje praktická ukázka z třídy OWLApiClass z balíčku OWLApi, kde pomocí metody *getDisjTridy* obdržíme všechny disjunktnivní třídy námi vybrané třídy.

Příklad 24: Naplnění listu disjunktivními třídami.

```
public ArrayList<OWLApiClass> getDisjTridy()

{

            ArrayList<OWLApiClass> disClassArray = new
ArrayList<OWLApiClass>();

            OWLClass myClass =
OWLApiBase.factory.getOWLClass(this.instance.getIRI());

            for (OWLClass disClass :
OWLApiBase.reasoner.getDisjointClasses(myClass).getFlattened())

      {
```

```
                  OWLApiClass myApiClass =
OWLApiBase.getOWLApiClass(disClass.getIRI());
            if (myApiClass == null)
            {
                  myApiClass = new
OWLApiClass(OWLApiBase.renderer.render(disClass), disClass, null);
            }
            disClassArray.add(myApiClass);
        }
      return disClassArray;
}
```

5.1.3.3 Ekvivalentní třída

Vlastnost ekvivalentních tříd se používá k označení, že dvě třídy mají přesně stejné instance.

Tento slovní popis lze zachytit takto:

Axiom	Podmínka
EquivalentClasses ($CE_1 \dots CE_n$)	$(CE_j)^C = (CE_k)^C$ pro každé $1 \le j \le n$ a pro každé $1 \le k \le n$

kde *EquivalentClasses* označuje ekvivalentní třídu. Symbol *CE* označuje *Class Expression*, tedy danou třídu a symbol *C* označuje *Class,* třídu.

Následuje praktická ukázka z třídy OWLApiClass z balíčku OWLApi, kde pomocí metody *getEkviTridy* obdržíme všechny ekvivalentní třídy námi vybrané třídy.

Příklad 25: Naplnění seznamu ekvivalentními třídami.

```
public ArrayList<OWLApiClass> getEkviTridy() {
            ArrayList<OWLApiClass> ekviClassArray = new
ArrayList<OWLApiClass>();
            OWLClass myClass =
OWLApiBase.factory.getOWLClass(this.instance.getIRI());
            Set<OWLEquivalentClassesAxiom> app = null;
            for (OWLClass ekviClass :
OWLApiBase.reasoner.getEquivalentClasses(myClass) )
      {
                  OWLApiClass myApiClass =
OWLApiBase.getOWLApiClass(ekviClass.getIRI());
            if (myApiClass == null)
            {
                  myApiClass = new
OWLApiClass(OWLApiBase.renderer.render(ekviClass), ekviClass, null);
            }
            ekviClassArray.add(myApiClass);
            app =
OWLApiBase.ont.getEquivalentClassesAxioms(myApiClass.instance);
      }
            return ekviClassArray;
```

```
}
```

5.1.3.4 Nadtřída (SuperClass)

Chápání pojmu Třída v OWL není odlišná od objektově orientovaného principu. Jedná se o soubor jedinců se stejnými vlastnostmi. Jsou organizovány do hierarchie - taxonomie tj. nadtříd a podtříd (**SuperClasses** a SubClasses). Vytvořením hierarchie tříd dodáváme naší ontologii strukturu.

Následuje praktická ukázka z třídy OWLApiClass z balíčku OWLApi, kde pomocí metody *getSuperTrida* obdržíme všechny nadtřídy námi vybrané třídy.

Příklad 26: Naplnění seznamu nadtřídami námi vybrané třídy.

```
public ArrayList<OWLApiClass> getSuperTrida() {
            ArrayList<OWLApiClass> supClassArray = new
ArrayList<OWLApiClass>();
            OWLClass myClass =
OWLApiBase.factory.getOWLClass(this.instance.getIRI());
            for (Node<OWLClass> supClass :
OWLApiBase.reasoner.getSuperClasses(myClass, true))
      {
                  OWLApiClass myApiClass =
OWLApiBase.getOWLApiClass(supClass.getRepresentativeElement().getIRI());
            if (myApiClass == null)
            {
                  myApiClass = new
OWLApiClass(OWLApiBase.renderer.render(supClass.getRepresentativeElement(
)), supClass.getRepresentativeElement(), null);
            }
            supClassArray.add(myApiClass);
      }
      return supClassArray;
}
```

5.1.3.5 Komentáře

Pomocí anotací, získáme jednotlivé komentáře k dané třídy. Sledovanou entitu můžeme specifikovat pomocí URI.

Následuje praktická ukázka z třídy OWLApiClass z balíčku OWLApi, kde pomocí metody *getCommenty* obdržíme všechny komentáře námi vybrané třídy.

```
public ArrayList<String> getCommenty() {
            ArrayList<String> commentyClassArray = new
ArrayList<String>();
            OWLClass commentyClass =
OWLApiBase.factory.getOWLClass(this.instance.getIRI());
            for (OWLAnnotation personAno :
commentyClass.getAnnotations(OWLApiBase.ont))
      {
      commentyClassArray.add(OWLApiBase.renderer.render(personAno.getValu
e()));
```

```
      }
      return commentyClassArray;
}
```

5.1.3.6 Individua třídy

Chceme-li získat individua, které jsou instance třídy využijeme k tomu metodu *getIndividua*.

Následuje praktická ukázka z třídy OWLApiClass z balíčku OWLApi, kde pomocí metody *getIndividua* obdržíme všechny individua námi vybrané třídy.

Příklad 27: Naplnění seznamu individuí.

```
public ArrayList<OWLApiIndividuals> getIndividua() {
            ArrayList<OWLApiIndividuals> individualClassArray = new
ArrayList<OWLApiIndividuals>();
            OWLClass individualClass =
OWLApiBase.factory.getOWLClass(this.instance.getIRI());
            for (OWLNamedIndividual individual :
OWLApiBase.reasoner.getInstances(individualClass, true).getFlattened())
            {
                  OWLApiIndividuals myApiIndividual =
OWLApiBase.getOWLApiIndividual(individual.getIRI());
                  if (myApiIndividual == null)
                  {
                        myApiIndividual = new OWLApiIndividuals();
                        myApiIndividual.setNazev(OWLApiBase.renderer.rend
er(individual));
                        myApiIndividual.setRodic(this);
                        myApiIndividual.setInstance(individual);
                  }
                  individualClassArray.add(myApiIndividual);
            }
              return individualClassArray;
}
```

5.1.4 Práce s individui

Obdobně jako v kapitole 5.1.3, si v této kapitole popíšeme, jak získat jednotlivé vlastnosti individuí z námi načtené OWL ontologie.

5.1.4.1 Typy individuí

Pomocí metody *getTypes*, dostaneme datové typy specifikovaného individua, které odpovídají datovému typu axiomu námi specifikované OWL ontologie.

Datový typ v OWL ontologii, můžeme definovat následovně:

Axiom	Podmínka
DatatypeDefinition (DT DR)	$(DT)^{DT} = (DR)^{DR}$

kde *DatatypeDefinition* označuje definici datového typu. Symbol *DT* označuje *Data Type*, tedy datový typ a symbol *DR* označuje *Data Range* – oblast dat.

Následuje praktická ukázka z třídy OWLApiClass z balíčku OWLApi, kde pomocí metody *getTypes* obdržíme všechny datové typy individuí námi vybrané třídy.

Příklad 28: Naplnění seznamu datovými typy.

```
public ArrayList<OWLApiClass> getTypes() {
            types = new ArrayList<OWLApiClass>();
            for (OWLClassExpression type :
this.getInstance().getTypes(OWLApiBase.ont))
            {
                  OWLApiClass myApiClass =
OWLApiBase.getOWLApiClass(type.asOWLClass().getIRI());
            if (myApiClass == null)
            {
                  myApiClass = new
OWLApiClass(OWLApiBase.renderer.render(type.asOWLClass()),
type.asOWLClass(), null);
            }
                  types.add(myApiClass);
            }
      return types;
}
```

5.1.4.2 Stejné individua

Pomocí metody *getSameIndividuals* můžeme dostat individua, která jsou stejné jako sledované individuum. Jako parametr předáme metodě individuum, které má být porovnáno.

Ekvivalentní individua v OWL ontologii, můžeme definovat následovně:

Axiom	Podmínka
SameIndividual (a_1 ... a_n)	$(a_j)^I = (a_k)^I$ pro každé $1 \leq j \leq n$ a pro každé $1 \leq k \leq n$

kde *SameIndividual* označuje ekvivalentní individua. Symbol *a* označuje dané individuum a symbolem *I* označíme sadu individuí.

Následuje praktická ukázka z třídy OWLApiClass z balíčku OWLApi, kde pomocí metody *getIndividual* obdržíme všechny ekvivalentní individuí s námi vybraným individuem.

Příklad 29: Naplnění seznamu ekvivalentními individui.

```
public ArrayList<OWLApiIndividuals> getSameIndividuals() {
            sameIndividuals = new ArrayList<OWLApiIndividuals>();
            for (OWLIndividual ind :
this.getInstance().getSameIndividuals(OWLApiBase.ont))
            {
            OWLApiIndividuals mySameIndiv =
OWLApiBase.getOWLApiIndividual(ind.asOWLNamedIndividual().getIRI());
            sameIndividuals.add(mySameIndiv);
            }
```

```
      return sameIndividuals;
}
```

5.1.4.3 Rozdílné individua

Pomocí metody *getDifferentIndividuals* získáme individua, které se liší od uvedeného individua. Jako parametr předáme individuum, jehož rozdílné individua mají být vráceny.

Rozdílné individua v OWL ontologii, můžeme definovat následovně:

Axiom	Podmínka
DifferentIndividuals (a_1 ... a_n)	$(a_j)^I \neq (a_k)^I$ pro každé $1 \leq j \leq n$ a pro každé $1 \leq k \leq n$, kde $j \neq k$

kde *DifferentIndividual* označuje rozdílné individua. Symbol *a* označuje dané individuum a symbolem *I* označíme sadu individuí.

Následuje praktická ukázka z třídy OWLApiClass z balíčku OWLApi, kde pomocí metody *getDiferentIndividuals* obdržíme všechny rozdílná individua s námi vybraným individuem.

Příklad 30: Naplnění seznamu rozdílnými individui.

```
public ArrayList<OWLApiIndividuals> getDiferentIndividuals() {
      diferentIndividuals = new ArrayList<OWLApiIndividuals>();
      for (OWLIndividual ind :
this.getInstance().getDifferentIndividuals(OWLApiBase.ont))
      {
            OWLApiIndividuals myDifirentIndiv =
OWLApiBase.getOWLApiIndividual(ind.asOWLNamedIndividual().getIRI());
            sameIndividuals.add(myDifirentIndiv);
      }
      return diferentIndividuals;
}
```

5.1.4.4 Vlastnosti objektu

Pomocí metody *getObjectPropertyValues* získáme vlastnosti objektu individua a specifikované vlastnosti. Jako parametr této metodě předáme OWL ontologii a vlastnost individua, které mají být vráceny.

Vlastnosti objektu v OWL ontologii, můžeme definovat následovně:

Axiom	Podmínka
ObjectPropertyAssertion (OPE a_1 a_2)	$((a_1)^I, (a_2)^I \in (OPE)^{OP}$

kde *ObjectPropertyAssertion* označuje vlastnosti objektů. Symbol *a* označuje dané individuum a symbolem *I* označíme sadu individuí. OPE značí *Object property expression* – výraz vlastnosti objektu a OP označuje *Object Property* – vlastnost objektu.

Následuje praktická ukázka z třídy OWLApiClass z balíčku OWLApi, kde pomocí metody *getObjectPropertyassertions* obdržíme všechny vlastnosti objektu s námi vybraným individuem.

Příklad 31: Naplnění seznamu vlastnostmi objektu.

```
public ArrayList<OWLApiDataProperties> getObjectPropertyassertions() {
      objectPropertyassertions = new ArrayList<OWLApiDataProperties>();
```

```
      Iterator it =
this.getInstance().getObjectPropertyValues(OWLApiBase.ont).entrySet().ite
rator();
      while (it.hasNext())
      {
            Map.Entry pairs = (Map.Entry)it.next();
            OWLApiDataProperties myApiProperie =
OWLApiBase.getOWLApiProperie(((OWLObjectPropertyExpression)pairs.getKey()
).asOWLObjectProperty().getIRI());
            objectPropertyassertions.add(myApiProperie);
      }
      return objectPropertyassertions;
}
```

5.1.4.5 Negativní vlastnosti objektu

Pomocí metody *getNegativeObjectPropertyValues* testujeme vlastnosti objektu individua, zda-li nejsou přiřazeny k danému individuu. Jako parametr této metodě předáme OWL ontologii, vlastnost individua, které má být testováno. A také vlastnost, která má být testována.

Negativní vlastnost objektu v OWL ontologii, můžeme definovat následovně:

Axiom	Podmínka
NegativeObjectPropertyAssertion (OPE a_1 a_2)	$((a_1)^I, (a_2)^I \notin (OPE)^{OP}$

kde *NegativeObjectPropertyAssertion* označuje vlastnosti objektů. Symbol *a* označuje dané individuum a symbolem *I* označíme sadu individuí. OPE značí *Object property expression* – výraz vlastnosti objektu a OP označuje *Object Property* – vlastnost objektu.

Následuje praktická ukázka z třídy OWLApiClass z balíčku OWLApi, kde pomocí metody *getNegativeObjectPropertyassertions* obdržíme všechny vlastnosti objektu s námi vybraným individuem. Symbolem ∉ značíme, že prvky nenáleží dané množině.

Příklad 32: Naplnění seznamu negativními vlastnostmi objektu.

```
public ArrayList<OWLApiDataProperties>
getNegativeObjectPropertyassertions() {
      negativeObjectPropertyassertions = new
ArrayList<OWLApiDataProperties>();
      Iterator it =
this.getInstance().getNegativeObjectPropertyValues(OWLApiBase.ont).entryS
et().iterator();
      while (it.hasNext())
            {
                  Map.Entry pairs = (Map.Entry)it.next();
                  OWLApiDataProperties myApiProperie =
OWLApiBase.getOWLApiProperie(((OWLObjectPropertyExpression)pairs.getKey()
).asOWLObjectProperty().getIRI());
                  negativeObjectPropertyassertions.add(myApiProperie);
            }
      return negativeObjectPropertyassertions;
}
```

5.1.5 Práce s vlastnostmi

Obdobně jako v kapitole 5.1.3 a kapitole 5.1.4, si v této kapitole popíšeme, jak získat jednotlivé vlastnosti vlastností z námi načtené OWL ontologie.

5.1.5.1 Doména

Pomocí metody *getDomains* zkoumáme axiomy OWL ontologie v dané doméně. Jako parametr této metodě předáme OWL ontologii. Návratovou hodnotou je pak sada OWL tříd v dané doméně.

Doménové vlastnosti objektu OWL ontologie, můžeme definovat následovně:

Axiom	Podmínka
ObjectPropertyDomain (OPE CE)	$\forall x,y : (x, y) \in (OPE)^{OP} \notin (OPE)^{OP} \supset x \in (CE)^{C}$

kde *ObjectPropertyDomain* označuje doménové vlastnosti objektů. Symboly ∀ označuje obecný kvantifikátor. OPE značí *Object property expression* – výraz vlastnosti objektu a OP označuje *Object Property* – vlastnost objektu. CE – *Class Expression*, značí danou třídu. Symbol *C* reprezentuje *Class* - třídu. Symbolem ∈ značíme, že prvky náleží dané množině. Symbolem ∉ značíme, že prvky nenáleží dané množině. Symbol ⊃ značí implikaci.

Následuje praktická ukázka z třídy OWLApiDataProperties z balíčku OWLApi, kde pomocí metody *getDomains* obdržíme všechny doménové vlastnosti objektu v námi specifikované OWL ontologii.

Příklad 33: Naplnění seznamu domén v námi specifikované OWL ontologii.

```
public ArrayList<OWLApiClass> getDomains()
{
      domains = new ArrayList<OWLApiClass>();
      for (OWLClassExpression domain :
this.getInstance().getDomains(OWLApiBase.ont))
            {
            OWLApiClass myApiClass =
OWLApiBase.getOWLApiClass(domain.asOWLClass().getIRI());
            domains.add(myApiClass);
            }
      return domains;
}
```

5.1.5.2 Rozsah

Pomocí metody *getRanges* získáme rozsahy vlastností, které jsou definované v dané OWL ontologii. Jako parametr této metodě předáme OWL ontologii, ve které mají být prozkoumány axiomy. Návratovou hodnotou je pak rozsah těchto vlastností axiomů.

Doménové vlastnosti objektu OWL ontologie, můžeme definovat následovně:

Axiom	Podmínka
ObjectPropertyRange (OPE CE)	$\forall x,y : (x, y) \in (OPE)^{OP} \notin (OPE)^{OP} \supset x \in (CE)^{C}$

kde *ObjectPropertyRange* označuje doménové vlastnosti objektů. OPE značí *Object property expression* – výraz vlastnosti objektu a OP označuje *Object Property* – vlastnost objektu. Označení CE – *Class Expression*, značí danou třídu. Symbol *C* reprezentuje *Class* - třídu. Symbol ∀ označuje obecný kvantifikátor. Symbolem ∈ značíme, že prvky náleží dané množině. Symbol ⊃ značí implikaci.

Následuje praktická ukázka z třídy OWLApiDataProperties z balíčku OWLApi, kde pomocí metody *getRanges* obdržíme všechny doménové rozsahy objektu v námi specifikované OWL ontologii.

Příklad 34: Naplnění seznamu rozsahu v námi specifikované OWL ontologii.

```
public ArrayList<OWLApiClass> getRanges() {
      ranges = new ArrayList<OWLApiClass>();
      for (OWLClassExpression range :
this.getInstance().getRanges(OWLApiBase.ont))
      {
            OWLApiClass myApiClass =
OWLApiBase.getOWLApiClass(range.asOWLClass().getIRI());
            ranges.add(myApiClass);
      }
      return ranges;
}
```

5.1.5.3 Ekvivalentní vlastnosti

Pomocí metody *getEquivalentProperties* získáme ekvivalentní vlastnosti tím, že metoda zkoumá axiómy v námi zadané OWL ontologii. Jako parametr předáme metodě OWL ontologii, ve které chceme zkoumat ekvivalentní axiomy.

Ekvivalentní vlastnosti objektu OWL ontologie, můžeme definovat následovně:

Axiom	Podmínka
EquivalentObjectProperties (OPE_1 ... OPE_n)	$(OPE_j)^{OP} = (OPE_j)^{OP}$ pro každé $1 \leq j \leq n$ a pro každé $1 \leq k \leq n$

kde *EquivalentObjectProperties* označuje ekvivalentní vlastnosti objektů. OPE značí *Object property expression* – výraz vlastnosti objektu. Parametr OP označuje *Object Property* – vlastnost objektu.

Následuje praktická ukázka z třídy OWLApiDataProperties z balíčku OWLApi, kde pomocí metody *getEquivalentProperties* obdržíme všechny ekvivalentní vlastnosti v námi specifikované OWL ontologii.

Příklad 35: Naplnění seznamu ekvivalentními vlastnostmi v námi specifikované OWL ontologii.

```
public ArrayList<OWLApiDataProperties> getEquivalentProperties() {
      equivalentProperties = new ArrayList<OWLApiDataProperties>();
      for (OWLObjectPropertyExpression equivalentProperti :
this.getInstance().getEquivalentProperties(OWLApiBase.ont))
            {
            OWLApiDataProperties myApiProperie =
OWLApiBase.getOWLApiProperie(equivalentProperti.asOWLObjectProperty().get
IRI());
            equivalentProperties.add(myApiProperie);
            }
      return equivalentProperties;
}
```

5.1.5.4 Super vlastnosti

Pomocí metody *getSuperProperties* získáme nadřazené, nebo-li super vlastnosti. Metoda zkoumá axiómy v námi zadané OWL ontologii. Jako parametr předáme metodě OWL ontologii, ve které chceme zkoumat nadřazené axiomy.

Následuje praktická ukázka z třídy OWLApiDataProperties z balíčku OWLApi, kde pomocí metody *getSuperProperties* obdržíme všechny nadřazené vlastnosti v námi specifikované OWL ontologii.

Příklad 36: Naplnění seznamu nadřazenými vlastnostmi v námi specifikované OWL ontologii.

```
public ArrayList<OWLApiDataProperties> getSuperProperties() {
      superProperties = new ArrayList<OWLApiDataProperties>();
      for (OWLObjectPropertyExpression superProperti :
this.getInstance().getSuperProperties(OWLApiBase.ont))
      {
            OWLApiDataProperties myApiProperie =
OWLApiBase.getOWLApiProperie(superProperti.asOWLObjectProperty().getIRI()
);
            superProperties.add(myApiProperie);
      }
      return superProperties;
}
```

5.1.5.5 Inversní vlastnosti

Pomocí metody *getInverses* získáme vlastnosti, které korespondují inverzním vlastnostem námi zkoumaných vlastností. Metoda zkoumá axiómy v námi zadané OWL ontologii. Jako parametr předáme metodě OWL ontologii, ve které chceme zkoumat inverzní vlastnosti.

Inversní vlastnosti objektu OWL ontologie, můžeme definovat následovně:

Axiom	Podmínka
InverseObjectProperties (OPE_1 OPE_2)	$(OP_1)^{OP} = \{(x, y) \mid (y,x)\} \in (OP_2)^{OP}$

kde *InverseObjectProperties* označuje inversní vlastnosti objektů. Výraz OP označuje *Object Property* – vlastnost objektu. Symbolem ∈ značíme, že prvky náleží dané množině.

Následuje praktická ukázka z třídy OWLApiDataProperties z balíčku OWLApi, kde pomocí metody *getInverses* obdržíme všechny inverzní vlastnosti v námi specifikované OWL ontologii.

Příklad 37: Naplnění seznamu inversními vlastnostmi v námi specifikované OWL ontologii.

```
public ArrayList<OWLApiDataProperties> getInverses() {
      inverses = new ArrayList<OWLApiDataProperties>();
      for (OWLObjectPropertyExpression invers :
this.getInstance().getInverses(OWLApiBase.ont))
      {
            OWLApiDataProperties myApiProperie =
OWLApiBase.getOWLApiProperie(invers.asOWLObjectProperty().getIRI());
            inverses.add(myApiProperie);
      }
      return inverses;
```

```
}
```

5.1.5.6 Sub vlastnosti

Pomocí metody *getSubProperties* získáme vlastnosti, které korespondují podřazeným vlastnostem námi zkoumaných vlastností. Metoda zkoumá axiómy v námi zadané OWL ontologii. Jako parametr předáme metodě OWL ontologii, ve které chceme zkoumat podřazené vlastnosti.

Podřazené vlastnosti objektu OWL ontologie, můžeme definovat následovně:

Axiom	Podmínka
SubObjectPropertyOf (OPE_1 OPE_2)	$(OPE_1)^{OP} \subseteq (OPE_2)^{OP}$

kde *InverseObjectProperties* označuje inversní vlastnosti objektů. Výraz OP označuje *Object Property* – vlastnost objektu. OPE značí *Object property expression* – výraz vlastnosti objektu. Symbolem ∈ značíme, že prvky náleží dané množině. Symbolem ⊆ značíme podmnožinu.

Následuje praktická ukázka z třídy OWLApiDataProperties z balíčku OWLApi, kde pomocí metody *getSubProperties* obdržíme všechny podřazené vlastnosti v námi specifikované OWL ontologii.

Příklad 38: Naplnění seznamu podřazených vlastnostmi v námi specifikované OWL ontologii.

```
public ArrayList<OWLApiDataProperties> getSubProperties() {

      subProperties = new ArrayList<OWLApiDataProperties>();

      for (OWLObjectPropertyExpression subProperti :
this.getInstance().getSubProperties(OWLApiBase.ont))

            {

            OWLApiDataProperties myApiProperie =
OWLApiBase.getOWLApiProperie(subProperti.asOWLObjectProperty().getIRI());

            subProperties.add(myApiProperie);

            }

      return subProperties;

}
```

5.1.5.7 Disjoint vlastnosti

Pomocí metody *getDisjointProperties* získáme vlastnosti zkoumaného axiomu, které korespondují prázdné množině dvou zkoumaných axiomů. Metoda zkoumá axiómy v námi zadané OWL ontologii. Jako parametr předáme metodě OWL ontologii, ve které chceme zkoumat vlastnosti axiomů.

Disjoint vlastnosti objektů OWL ontologie, můžeme definovat následovně:

Axiom	Podmínka
DisjointObjectProperties (OPE_1 ... OPE_n)	$(OPE_j)^{OP} \cap (OPE_k)^{OP} = \emptyset$ pro každé $1 \le j \le n$ a pro každé $1 \le k \le n$, kde $j \ne k$

kde *DisjointObjectProperties* označuje disjoint vlastnosti objektů. Výraz OP označuje *Object Property* – vlastnost objektu. OPE značí *Object property expression* – výraz vlastnosti objektu. Symbolem ∅ označujeme prázdnou množinu. Symbolem ∩ označujeme průnik množin.

Následuje praktická ukázka z třídy OWLApiDataProperties z balíčku OWLApi, kde pomocí metody *getSubProperties* obdržíme všechny podřazené vlastnosti v námi specifikované OWL ontologii.

Příklad 39: Naplnění seznamu disjoint vlastnostmi v námi specifikované OWL ontologii.

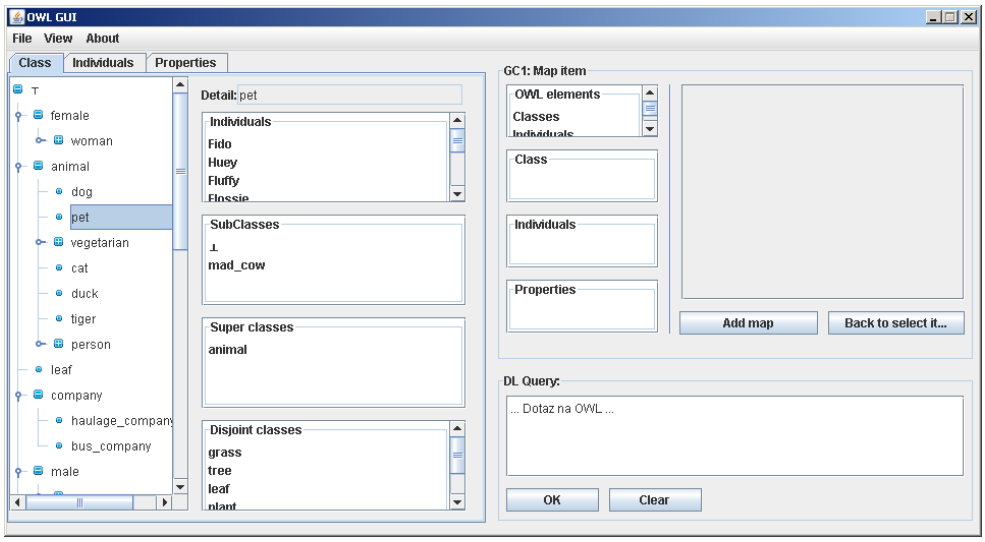
```
public ArrayList<OWLApiDataProperties> getDisjointProperties() {

      disjointProperties = new ArrayList<OWLApiDataProperties>();

      for (OWLObjectPropertyExpression disjointProperti :
this.getInstance().getDisjointProperties(OWLApiBase.ont))

      {

            OWLApiDataProperties myApiProperie =
OWLApiBase.getOWLApiProperie(disjointProperti.asOWLObjectProperty().getIR
I());

            disjointProperties.add(myApiProperie);

      }

      return disjointProperties;

}
```

5.2 Návrh grafického rozhraní

Celá kapitola 5.1 popisovala jak načíst vstupní OWL ontologii do jádra Frameworku. Dalším krokem je tedy reprezentace načtených axiómu v grafickém uživatelském prostředí. Samotné vykreslení OWL ontologie a zobrazení dle uživatelem zvolených grafických prvků, není předmětem téhle diplomové práce. Tato funkcionalita proto není ve výsledném Frameworku naprogramována. Taktéž modifikace a zpětné uložení OWL ontologie do OWL/XML souboru je předmětem jiné diplomové práce.

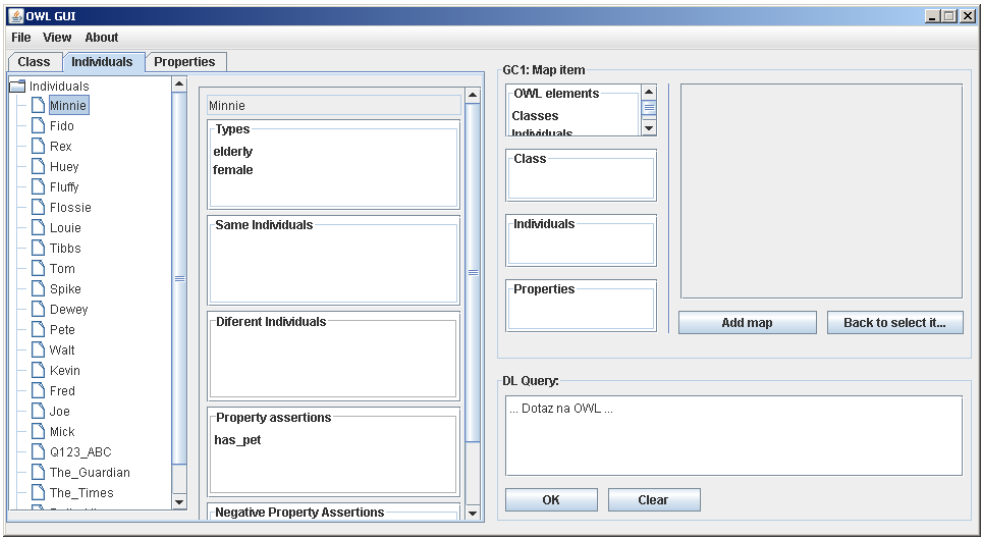
Na obrázku 8 je ukázka výsledného grafického rozhraní pro práci s třídami. Hlavní okno je pomyslně rozděleno na dvě hlavní části. V levé části, lze procházet načtenou OWL ontologií. Lze si zobrazovat jednotlivé vlastnosti tříd, individuí a vlastností. V pravé části uživatelského rozhraní je předpřipravena pro mapování axiomů na grafické komponenty.

Následuje ukázka jednotlivých oken uživatelského rozhraní pro zobrazení tříd, individuí a vlastností nad načtenou OWL ontologií.



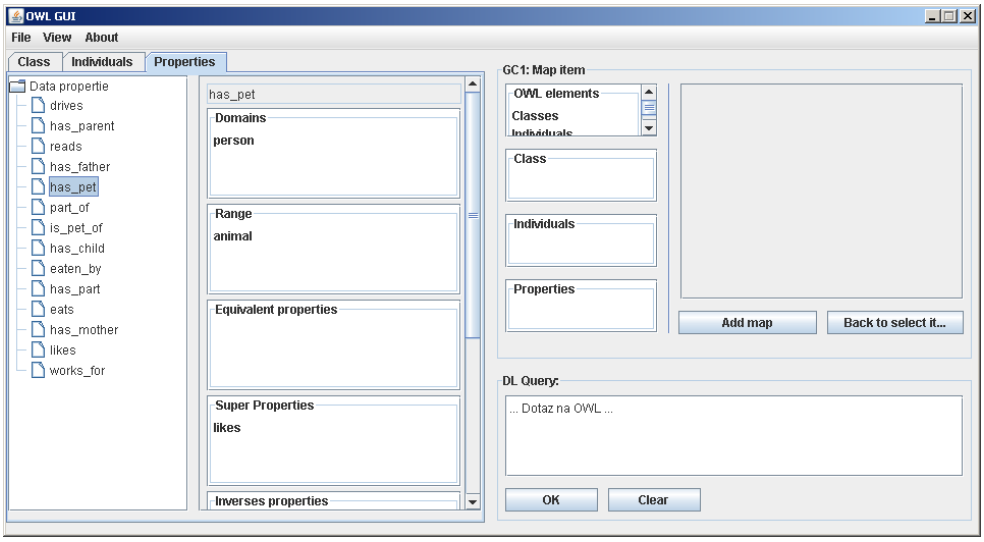
Obrázek 8: Náhled GUI a stromová struktura tříd z načtené OWL ontologie.

Na obrázku 9 je ukázka výsledného grafického rozhraní pro práci s individui.



Obrázek 9: Náhled GUI a stromová struktura individuí z načtené OWL ontologie.

Na obrázku 10 je ukázka výsledného grafického rozhraní pro práci s vlastnostmi.



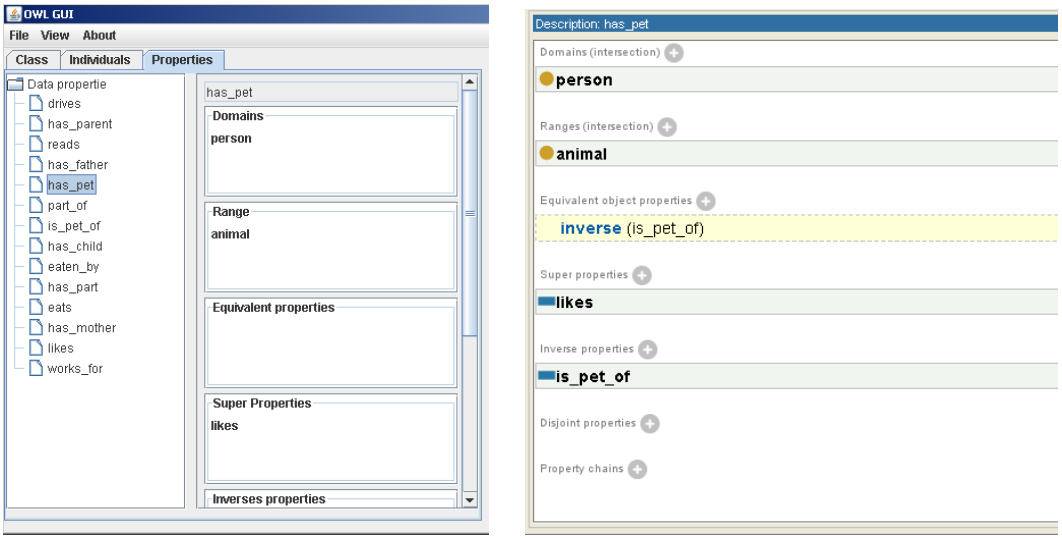
Obrázek 10: Náhled GUI a stromová struktura vlastnosti z načtené OWL ontologie.

6 Testování

Správnost implementace metod pro načtení OWL ontologie do jádra aplikace jsem průběžně testoval v kapitole 1 popsaném open source SW *Protégé* [*3*]. Pro testovací účely jsem využil *Protégé* ve verzi 4.1. Výchozí *Reasoner* jsem zvolil *FaCT++,* volně dostupný z [*22*]. Naproti tomu v naprogramovaném Frameworku používám jako výchozí *Reasoner HermiT.* Ověřil jsem si tak správnost naprogramovaných metod.

Jako vstupní OWL ontologii jsem jednak využil volně dostupnou ontologii *People,* dostupnou z [*15*]. Dále za účelem testováni správné funkcionality jsem vytvořil ontologii Manželství ve formátu OWL/XML, která je dostupná na přiloženém CD. Při postupném testovaní všech naimplementovaných metod jsem dostal na výstupu stejné výsledky, jako výsledky v SW *Protégé 4.1*.

Jako ukázku přikládám výstup z Frameworku OWL API a výstup z *Protégé 4.1*. Vlastnost *has_pet* patří do domény *person,* rozsah axiomu je *animal.* Vlastnost *has_pet* má nad vlastnost *likes.* Inverzní vlastnost je *is_pet_of.*



Obrázek 11: Srovnání výstupu vlastnosti *has_pet* ve Frameworku OWL API a Protégé 4.1.

7 Závěr

Cílem této diplomové práce bylo vytvořit Framework pro načtení vstupní OWL ontologie do jádra aplikace. Za tímto účelem, bylo nezbytně nutné, plně porozumět samotnému jazyku OWL ve verzi alespoň DL. Po zvládnutí syntaxe samotného jazyka, jsem se zaměřil na další část diplomové práce a to jakým způsobem načíst vstupní ontologii do jádra aplikace. Bylo nutné úzce spolupracovat s dalším diplomantem, neb výstup mojí práce bude vstupem pro diplomovou práci grafického zobrazení načtené ontologie. Po vyzkoušení několika knihoven pro práci s XML soubory, se jako nejvhodnější nástroj osvědčila sada knihoven OWL API. Dalším úkolem, který se mi podařilo splnit, bylo nastudovat, jak pracovat s těmito knihovnami a efektivně je využít ve vytvářeném Frameworku. Výsledkem diplomové práce je připravený Framework, který může být dál rozšířen o funkcionalitu vykreslení načtené ontologie. Také jsem připravil první verzi grafického uživatelského rozhraní, kde je možné vizuálně procházet načtenou OWL ontologii. Jak však již bylo zmíněno výše, samotná grafická stránka aplikace je předmětem jiné diplomové práce.

Přínosem této diplomové práce je bezpochyby i textová část, kde je shrnuta teoretická část k ontologiím. Následuje teorie k OWL syntaxi. Nejdůležitější část textové části, však shledávám v kapitole 5 – samotné řešení Frameworku. Tuto kapitolu lze využít jako tutoriál, jak si naimplementovat vlastní prohlížeč ontologií. Jelikož dostupné zdroje nejsou téměř v našem rodném jazyce, může budoucím čtenářům posloužit tato kapitola jako průvodce pro jejich vlastní řešení. Naprogramovaný Framework je obsahem přiloženého CD a tak spojením těchto dvou zdrojů nebude problém pro budoucí čtenáře plnit pochopit funkcionalitu naprogramovaného Frameworku.

Danou problematikou se také zabývají lidé z Manchesterské veřejné univerzity a tak jak již bylo napsáno, dostupné materiály jsou povětšinou v anglickém jazyce. Na druhou stranu je však nutné uvést, že dostupný materiál je velmi kvalitně zpracován. Lze tedy doporučit sadu knihoven OWL API jako dobře zpracovaný interface pro práci s OWL ontologiemi.

8 Literatura

[1] *WebOnto* [online]. 2011 [cit. 2011-12-11]. Dostupné na URL: <http://projects.kmi.open.ac.uk/webonto/>

[2] *OntoEdit* [online]. 2011 [cit. 2010-12-11]. Dostupné na URL: <http://www.hanyas.net/seweb/editor.php?article=105>

[3] *Protegé* [online]. [cit. 2010-12-11]. Dostupné na URL: <http://protege.stanford.edu>

[4] SVÁTEK, Vojtěch. *Ontologie a WWW* [online]. Brno : Dakaton, 2002. 35 s. Oborová práce. VŠE Praha, Katedra informačního a znalostního inženýrství. Dostupné na URL: <http://nb.vse.cz/~svatek/onto-www.pdf>.

[5] USCHOLD, M., GRUNINGER, M.: Ontologies: principles, methods and applications. The Knowledge Engineering Review, Vol.11:2, 1996

[6] LACY, L. W. OWL: Representing Information Using the Web Ontology Language. ISBN 141203448-5

[7] SVÁTEK, Vojtěch a VACURA, Miroslav. Ontologické inženýrství. In *DATAKON 2007: sborník databázové konference*. Brno, Česká republika 20.-23. října 2007. Brno: Masarykova univerzita, 2007. [cit. 2012-04-20]. Dostupné na URL: <http://nb.vse.cz/~svatek/dkon07final.pdf>

[8] *Ontology definition metamodel (ODM)* [online]. Version 1.0. OMG Document Number: formal/2009-05-01. Object Management Group, May 2009 [cit. 22.4.2012], s. 31. Dostupné na URL: <http://www.omg.org/spec/ODM/1.0>

[9] DE NICOLA, Antonio, MISSIKOFF, Michele a NAVIGLI, Roberto. A software engineering approach to ontology building. *Information systems*. 2009, vol. 34, no. 2 (April), s. 260. ISSN 0306-4379. DOI 10.1016/j.is.2008.07.002

[10] GOMEZ-PEREZ, A., FERNANDEZ-LOPEZ, M., CORCHO, O.*: Ontological Engineering: with Examples from the Areas of Knowledge Management, E-Commerce and the Semantic Web.* Springer, 2003. ISBN: 9781846283963 1846283965

[11] *The University of Manchester* [online]. 2011 [cit. 2011-12-11]. Dostupné na URL: <http://www.manchester.ac.uk/>

[12] SourceForge: *OWL API* [online]. 2012 [cit. 2012-04-20]. Dostupné na URL: <http://owlapi.sourceforge.net/documentation.html>

[13] MarketPlace [online]. 2012 [cit. 2012-04-20]. Dostupné na URL: <http://marketplace.eclipse.org/content/subversive-svn-team-provider>

[14] HermiT [online]. 2012 [cit. 2012-04-20]. Dostupné na URL: <http://hermit-reasoner.com>

[15] OWL Ontologie People [online]. 2012 [cit. 2012-04-22]. Dostupné na URL: <http://owl.man.ac.uk/2005/07/sssw/people>

[16] ŠTOLFA, Svatopluk, KOŽUSZNIK, Jan. Knowledge based approach to software development process modeling. 2011, ISBN: 978-364222409-6.

[17] W3C (2009) OWL 2 Web Ontology Language. 2012 [cit. 2012-04-22]. Dostupné na URL: <http://www.w3.org/TR/owl2-overview/>

[18] DUŽÍ, Marie. Matematické logika (Mgr). 2010. [cit. 2012-04-22]. Dostupné na URL: <http://www.cs.vsb.cz/duzi/>

[19] HEROUT, Pavel. Učebnice jazyka Java (Java 5). 2008, ISBN: 978-80-7232-355-5. EAN: 9788072323555.

[20] HEROUT, Pavel. Java - grafické uživatelské prostředí a čeština. 2008, ISBN: 80-7232-328-9

[21] SourceForge: OWL API [online]. 2012 [cit. 2012-04-22]. Dostupné na URL: <http://owlapi.sourceforge.net/javadoc/overview-summary.html >

[22] *The University of Manchester.* FaCT++. [online]. 2007 [cit. 2012-04-22]. Dostupné na URL: <http://owl.man.ac.uk/factplusplus/>

[23] *Znalostní technologie I* [online]. 2009 [cit. 2012-04-22]. Dostupné na URL: <http://lide.uhk.cz/fim/ucitel/fshusam2/lekarnicky/zt1/zt1_kap04.html#owluvod>

9 Obsah přiloženého CD

- Diplomová práce
- Abstrakt
- Framework
- Testovací ontologie